Chapter 4.3: Inheritance

Object Oriented Programming with Java

Superclass and subclass | extends | super and constructor chaining

Overriding and @Override | protected | final | is-a and has-a

Inheritance or composition? | Multiple inheritance | The Object class

toString, equals, hashCode

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_03_Inheritance.html

INTRODUCTION

Learning Outcomes

Concept

Define inheritance, superclass and subclass; recognize the “is-a” relationship.

Syntax

Write a subclass with `extends`; explain what is inherited and what is not.

Constructors

Use `super(...)` and `super.method()`; trace the order of the constructor chain.

Overriding

Override a method, tell overriding from overloading, and use `@Override`.

Access and restriction

Explain `protected` access and how `final` restricts inheritance.

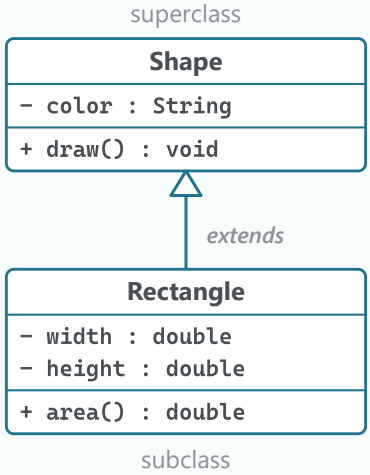
The Object class

Write `toString`, `equals` and `hashCode` correctly; know the difference between `==` and `equals`.

What Is Inheritance?

- **Inheritance**: building a **new class** from an existing one by taking over its fields and methods and adding new or modified behavior.
- The existing class: the **superclass**.
- The new class: the **subclass**.
- A subclass can add fields and methods; inherited fields are **not redeclared**.

Analogy: “Every rectangle is a shape.” A rectangle has everything a shape has, plus its own `width`, `height` and `area()`.

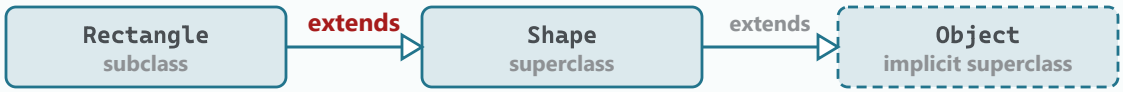


Subclassing with extends

```java
// base class (superclass)
public class Shape {
    // fields and methods of the base class
    // ...
}

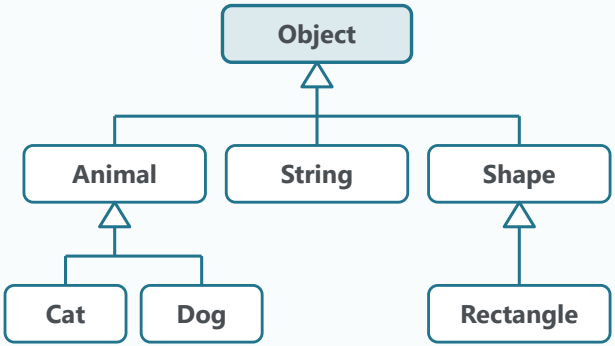
// new class (subclass), derived from Shape
public class Rectangle extends Shape {
    // fields and methods of the new class
    // ...
}
```

- Syntax: `class Sub extends Super`. `extends` means “builds on”.
- `extends` names **exactly one** class.
- The subclass uses the superclass’s `public` and `protected` members as if they were its own.
- Redeclaring a superclass field with the same name in the subclass **hides** it: you get two separate variables and a lot of confusion. Don’t do it.
- A class without `extends` implicitly extends `Object`.



Subclass and Superclass

- A subclass behaves like its superclass and **adds** behavior of its own.
- This is why inheritance is also called **specialization**: the subclass represents a more **specific** group.
- Every cat is an animal, but not every animal is a cat.
- The Java class hierarchy starts with the `Object` class in package `java.lang`.
- Every class inherits from `Object`, **directly** or **indirectly**.

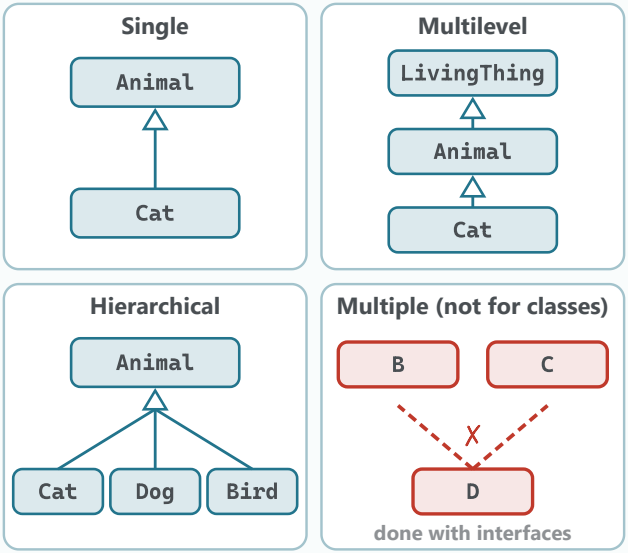


up: general · down: specific

Kinds of Inheritance

- **Single**: one subclass, one superclass. `Cat extends Animal`.
- **Multilevel**: a chain. `Cat` gets the members of `LivingThing` through `Animal`; each level carries everything of the one above.
- **Hierarchical**: many subclasses of one superclass. Common code is written once; each subclass adds only its differences.
- **Multiple**: a class with several superclasses. Java does not allow this for classes; interfaces cover the same need.

In Java the class hierarchy is a **tree**: its root is `Object`, and every class except `Object` has exactly **one** direct superclass. Chains can be arbitrarily deep, but deep chains are hard to read and change.



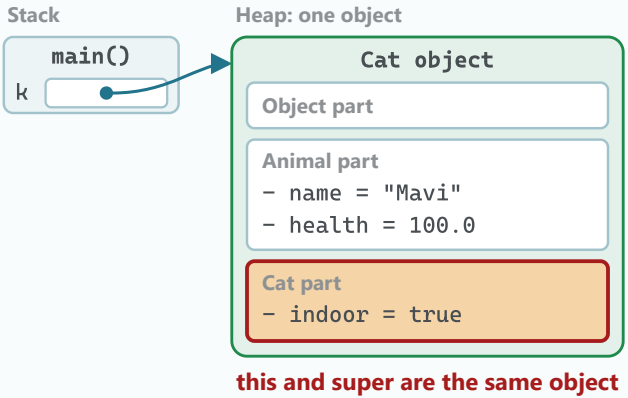
What Is Inherited?

Member	Inherited by the subclass?	Note
`public` / `protected` fields and methods	**Yes**	Used directly in the subclass.
Members with no modifier (package access)	Only within the same package	A subclass in another package cannot access them.
`private` fields	**Present** in the object, but not directly accessible	Used through the superclass's `public` / `protected` methods (getters, setters).
`private` methods	No	Invisible; cannot be overridden.
Constructors	**No**	Not inherited; the subclass constructor calls them with `super(...)`.
`static` members	Accessible	Belong to the class; not overridden, only hidden.

In short: a subclass object contains all the superclass fields; the access modifier decides which ones it can reach **directly**.

Inside a Subclass Object

- `new Cat(...)` creates **one** object; there is no separate object for the superclass.
- The object also has room for the fields inherited from the superclass. They are there even when `private`; only `Cat`'s code cannot name them.
- `super` is not a separate object: it is a way to reach the superclass part of the same object through the superclass's code.



PROOF: `getClass()` IN THE CONSTRUCTOR

```
class Animal {
    Animal() {
        System.out.println("In Animal(): "
            + getClass().getSimpleName());
    }
}
class Cat extends Animal { }
// in main:
new Cat();
new Animal();
```

```
In Animal(): Cat
In Animal(): Animal
```

Even while the `Animal` constructor runs, the object's real class is `Cat`: the constructor chain builds the parts of the **same** object in turn.

Moving Common Code Up

Generalization: finding fields and methods repeated in several classes and moving them into a common superclass. It is the reverse of specialization.

BEFORE: THE SAME CODE IN TWO PLACES

```
class Student {
    private String name;
    private int age;
    private double gpa;
    String info() { return name + " (" + age + ")"; }
}
class Teacher {
    private String name;
    private int age;
    private String subject;
    String info() { return name + " (" + age + ")"; }
}
```

If the age format changes, both classes must be fixed; forget one and the output is inconsistent.

AFTER: THE COMMON PART LIVES IN `Person`

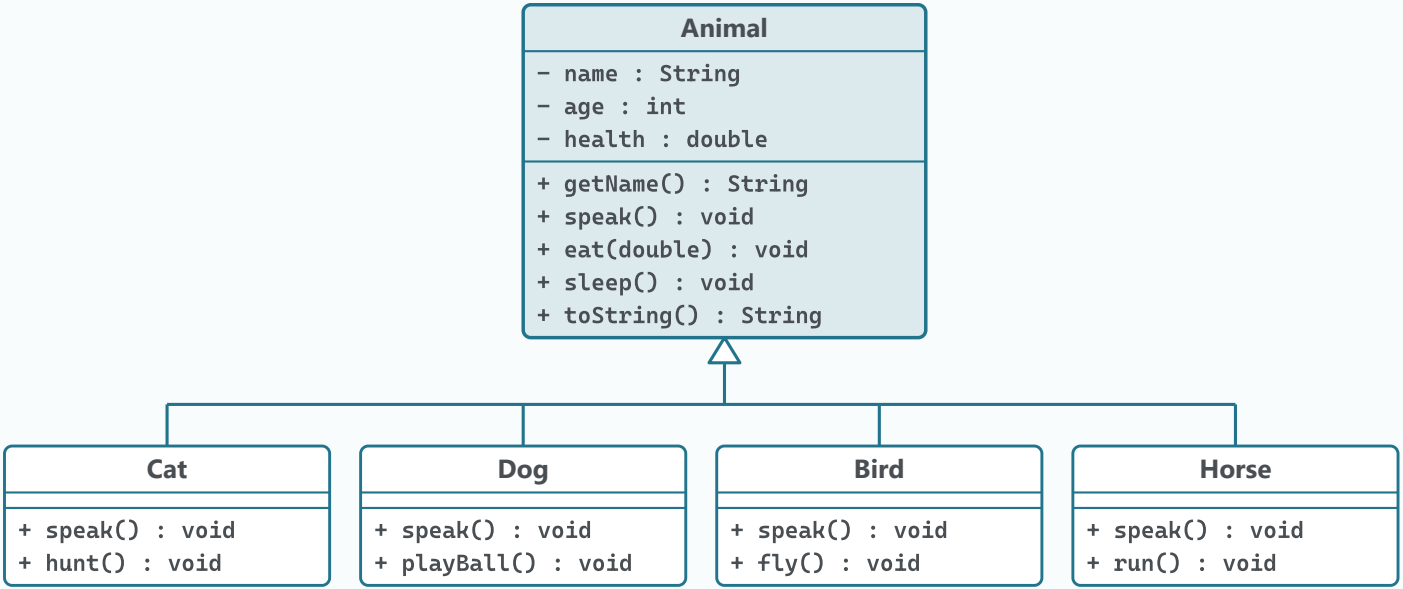
```
class Person {
    private String name;
    private int age;
    Person(String n, int a) { name = n; age = a; }
    String info() { return name + " (" + age + ")"; }
}
class Student extends Person {
    private double gpa;
    Student(String name, int age, double gpa) {
        super(name, age);
        this.gpa = gpa;
    }
    @Override String info() {
        return super.info() + ", gpa " + gpa;
    }
}   // Teacher likewise: adds subject
```

```
Ece (20), gpa 3.4
Murat (41), Physics
```

Ask before moving: “Is every student a person?” If yes, generalization is right; if the two classes merely share a few fields, composition fits better.

ANIMAL EXAMPLE

The Animal Class Hierarchy



Common fields and methods are written **once**, in `Animal`. Each subclass **overrides** `speak()` in its own way and adds a method of its own. In UML, inheritance is drawn as an arrow with a **hollow triangle** pointing to the superclass.

ANIMAL EXAMPLE

The Animal and Dog Classes

```
public class Animal {
    private String name;     // common fields
    private int age;
    private double health;

    public Animal(String name, int age) {
        this.name = name;
        this.age = age;
        this.health = 100.0;
    }
    public String getName() { return name; }   // access to private name
    public void speak() {
        System.out.println("This animal speaks.");
    }
    public void sleep() {
        health += 5;
        System.out.println(name + " slept. Health: " + health);
    }
}
```

Superclass `Animal`: the fields and behavior shared by all animals.

ANIMAL EXAMPLE

The Animal and Dog Classes

```
public class Dog extends Animal {
    public Dog(String name, int age) {
        super(name, age);   // superclass sets the fields
    }
    @Override
    public void speak() {   // changes inherited behavior
        System.out.println(getName() + " barks.");
    }
    public void playBall() { // behavior only Dog has
        System.out.println(getName() + " is playing ball!");
    }
}
```

```
Dog dog = new Dog("Karabaş", 4);
dog.speak();     // Dog's version
dog.playBall();   // Dog only
dog.sleep();       // inherited
```

```
Karabaş barks.
Karabaş is playing ball!
Karabaş slept. Health: 105.0
```

The subclass `Dog` writes only the **difference**; `sleep()` works without being written at all.

ANIMAL EXAMPLE

Benefits of Inheritance

Reuse

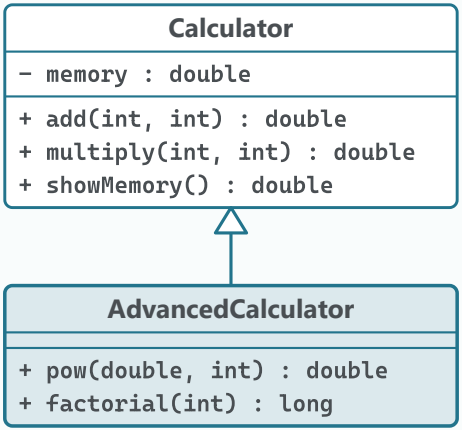
Superclass methods and fields are used in the subclass without being rewritten.

Extension

An existing class gains new features without changing its code.

Modification

Superclass methods can be changed in the subclass by **overriding** them.




```java
public class AdvancedCalculator
        extends Calculator {
    // new ability: exponentiation
    public double pow(double base, int pow) {
        double result = 1.0;
        for (int i = 0; i < pow; i++) {
            result *= base;
        }
        return result;
    }
    // factorial(int) is added likewise
}
// in main
AdvancedCalculator h = new AdvancedCalculator();
System.out.println(h.add(5, 3));         // 8.0  (inherited)
System.out.println(h.showMemory()); // 8.0  (inherited)
System.out.println(h.pow(2, 5));            // 32.0 (new)
```

The super Keyword

- `super(...)`: called inside a subclass constructor, it runs the **superclass constructor**, which initializes the superclass fields.
- `super(...)` must be the **first statement** of the constructor (before Java 25; Java 25 allows statements that do not touch the object before it).
- If omitted, the compiler silently inserts `super();` (no arguments).
- `super.method()`: reaches a superclass method with a dot; used to call the superclass version from inside an overriding method.
- Principle: **each class manages only its own data**. Cat does not assign Animal's `name` field itself; it leaves that to `super(name)`.

```
class Cat extends Animal {
    private boolean indoor;

    Cat(String name, boolean indoor) {
        super(name);          // Animal(String)
        this.indoor = indoor;  // its own data
    }

    @Override
    public String toString() {
        return super.toString()   // Animal's
               + ", Indoor: " + indoor;
    }
}
// in main
System.out.println(new Cat("Mavi", true));
```

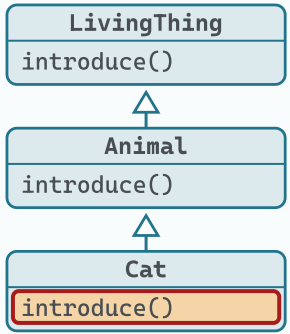
OUTPUT

```
Name: Mavi, Health: 100.0, Indoor: true
```

Animal's `toString()` returns the name and health; Cat appends only its own field.

The `super.introduce()` Chain: Code Trace

```
1  class LivingThing {
2      void introduce() { System.out.println("I'm
   alive"); }
3  }
4  class Animal extends LivingThing {
5      void introduce() {
6          super.introduce();
7          System.out.println("I'm an animal");
8      }
9  }
10 class Cat extends Animal {
11     void introduce() {
12         super.introduce();
13         System.out.println("I'm a cat");
14     }
15 }
▶  new Cat().introduce();   // main
```



Call stack

`Cat.introduce()`

`main()`

obj: Cat #1

`lookup starts at the real class: Cat.introduce()`

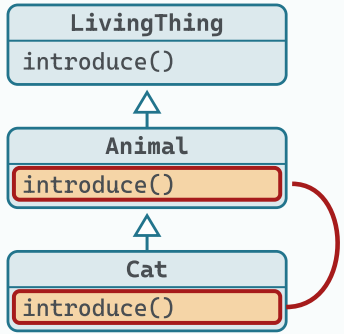
The object is a `Cat`; lookup starts at the bottom and finds `Cat.introduce()`.

Step 1 / 7

OUTPUT

The `super.introduce()` Chain: Code Trace

```
1  class LivingThing {
2      void introduce() { System.out.println("I'm
   alive"); }
3  }
4  class Animal extends LivingThing {
5      void introduce() {
6          super.introduce();
7          System.out.println("I'm an animal");
8      }
9  }
10 class Cat extends Animal {
11     void introduce() {
▶          super.introduce();
13         System.out.println("I'm a cat");
14     }
15 }
16 new Cat().introduce();   // main
```



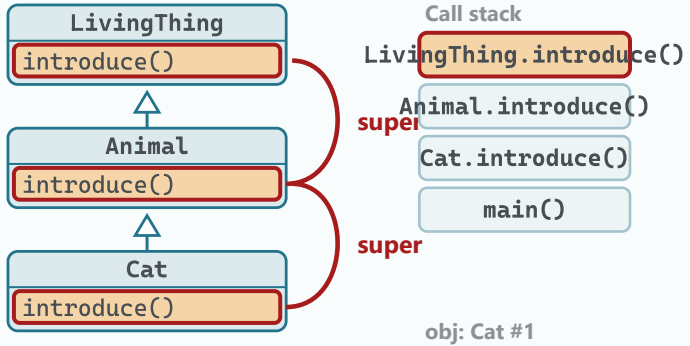
`super.introduce()` calls the version **one level up**. `Cat.introduce` waits on the stack.

Step 2 / 7

OUTPUT

The `super.introduce()` Chain: Code Trace

```
1  class LivingThing {
2      void introduce() { System.out.println("I'm
   alive"); }
3  }
4  class Animal extends LivingThing {
▶      void introduce() {
▶          super.introduce();
7          System.out.println("I'm an animal");
8      }
9  }
10 class Cat extends Animal {
11     void introduce() {
12         super.introduce();
13         System.out.println("I'm a cat");
14     }
15 }
16 new Cat().introduce();   // main
```



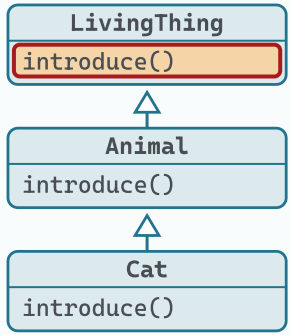
The object is still a `Cat`, yet `super` inside `Animal` always means `LivingThing`: **no** dynamic binding.

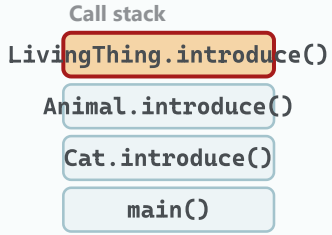
Step 3 / 7

OUTPUT

The `super.introduce()` Chain: Code Trace

```
1  class LivingThing {
▶      void introduce() { System.out.println("I'm
   alive"); }
3  }
4  class Animal extends LivingThing {
5      void introduce() {
6          super.introduce();
7          System.out.println("I'm an animal");
8      }
9  }
10 class Cat extends Animal {
11     void introduce() {
12         super.introduce();
13         System.out.println("I'm a cat");
14     }
15 }
16 new Cat().introduce();   // main
```





obj: Cat #1

I'm alive printed, LivingThing.introduce() returns

The topmost version runs and prints the first line, then returns.

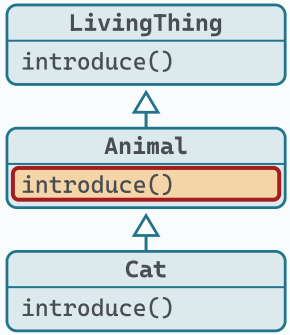
Step 4 / 7

OUTPUT

```
I'm alive
```

The `super.introduce()` Chain: Code Trace

```
1  class LivingThing {
2      void introduce() { System.out.println("I'm
   alive"); }
3  }
4  class Animal extends LivingThing {
5      void introduce() {
6          super.introduce();
▶          System.out.println("I'm an animal");
8      }
9  }
10 class Cat extends Animal {
11     void introduce() {
12         super.introduce();
13         System.out.println("I'm a cat");
14     }
15 }
16 new Cat().introduce();   // main
```



Call stack

- Animal.introduce()
- Cat.introduce()
- main()

obj: Cat #1

`Animal.introduce()` resumes where it left off: **I'm an animal**.

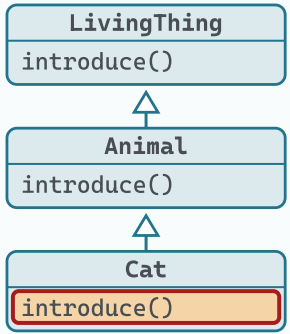
Step 5 / 7

OUTPUT

```
I'm alive
I'm an animal
```

The `super.introduce()` Chain: Code Trace

```
1  class LivingThing {
2      void introduce() { System.out.println("I'm
   alive"); }
3  }
4  class Animal extends LivingThing {
5      void introduce() {
6          super.introduce();
7          System.out.println("I'm an animal");
8      }
9  }
10 class Cat extends Animal {
11     void introduce() {
12         super.introduce();
▶          System.out.println("I'm a cat");
14     }
15 }
16 new Cat().introduce();   // main
```



Call stack

`Cat.introduce()`

`main()`

obj: Cat #1

Finally `Cat.introduce()` prints its own line.

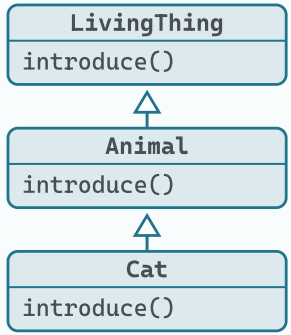
Step 6 / 7

OUTPUT

```
I'm alive
I'm an animal
I'm a cat
```

The `super.introduce()` Chain: Code Trace

```
1  class LivingThing {
2      void introduce() { System.out.println("I'm
   alive"); }
3  }
4  class Animal extends LivingThing {
5      void introduce() {
6          super.introduce();
7          System.out.println("I'm an animal");
8      }
9  }
10 class Cat extends Animal {
11     void introduce() {
12         super.introduce();
13         System.out.println("I'm a cat");
14     }
15 }
▶  new Cat().introduce();   // main
```



Call stack

main()

obj: Cat #1

calls **bottom-up**, output **top-down**

The calls went Cat → Animal → LivingThing and the output came back in reverse: each class first had its superclass do its work, then added its own difference.

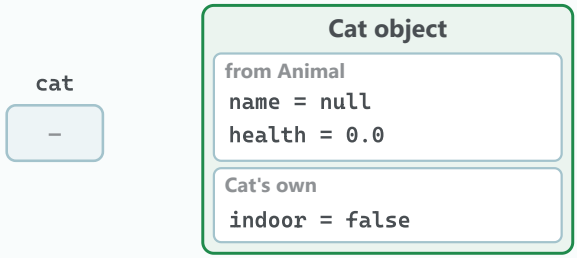
Step 7 / 7

OUTPUT

```
I'm alive
I'm an animal
I'm a cat
```

Constructor Chain: `new Cat("Mavi", true)`

```
1  class Animal {
2      private String name;
3      private double health;
4      Animal(String name) {
5          super();   // calls Object()
6          this.name = name;
7          health = 100.0;
8          System.out.println("Animal
   constructor");
9      }
10 }
11 class Cat extends Animal {
12     private boolean indoor;
13     Cat(String name, boolean indoor) {
14         super(name);
15         this.indoor = indoor;
16         System.out.println("Cat constructor");
17     }
18 }
19 // in main
▶  Cat cat = new Cat("Mavi", true);
21 System.out.println("Object ready");
```



`new` : memory for the object is allocated first; **all** fields (Animal's included) get default values.

Step 1 / 11

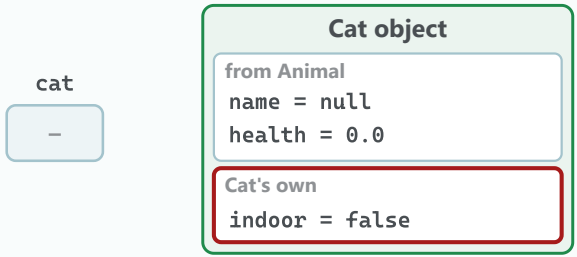
CALL STACK

`main()`

OUTPUT

Constructor Chain: `new Cat("Mavi", true)`

```
1  class Animal {
2      private String name;
3      private double health;
4      Animal(String name) {
5          super();   // calls Object()
6          this.name = name;
7          health = 100.0;
8          System.out.println("Animal
   constructor");
9      }
10 }
11 class Cat extends Animal {
12     private boolean indoor;
▶      Cat(String name, boolean indoor) {
14         super(name);
15         this.indoor = indoor;
16         System.out.println("Cat constructor");
17     }
18 }
19 // in main
20 Cat cat = new Cat("Mavi", true);
21 System.out.println("Object ready");
```



The Cat constructor is called. Before it does its own work, the superclass part must be ready.

Step 2 / 11

CALL STACK

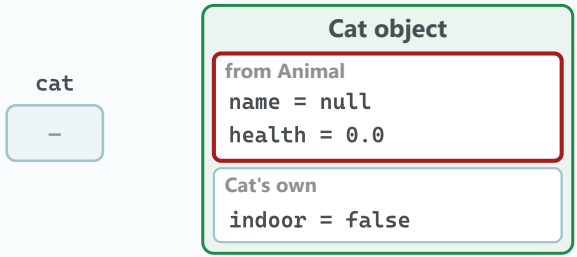
`Cat("Mavi", true)`

`main()`

OUTPUT

Constructor Chain: `new Cat("Mavi", true)`

```
1  class Animal {
2      private String name;
3      private double health;
4      Animal(String name) {
5          super();   // calls Object()
6          this.name = name;
7          health = 100.0;
8          System.out.println("Animal
   constructor");
9      }
10 }
11 class Cat extends Animal {
12     private boolean indoor;
13     Cat(String name, boolean indoor) {
▶          super(name);
15         this.indoor = indoor;
16         System.out.println("Cat constructor");
17     }
18 }
19 // in main
20 Cat cat = new Cat("Mavi", true);
21 System.out.println("Object ready");
```



`super(name)` : the Animal constructor is called; the Cat constructor waits.

Step 3 / 11

CALL STACK

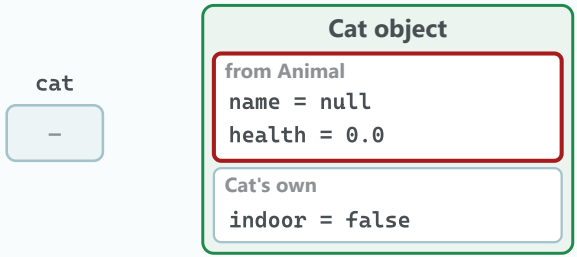
`Cat("Mavi", true)`

`main()`

OUTPUT

Constructor Chain: `new Cat("Mavi", true)`

```
1  class Animal {
2      private String name;
3      private double health;
4      Animal(String name) {
▶          super();   // calls Object()
6          this.name = name;
7          health = 100.0;
8          System.out.println("Animal
   constructor");
9      }
10 }
11 class Cat extends Animal {
12     private boolean indoor;
13     Cat(String name, boolean indoor) {
14         super(name);
15         this.indoor = indoor;
16         System.out.println("Cat constructor");
17     }
18 }
19 // in main
20 Cat cat = new Cat("Mavi", true);
21 System.out.println("Object ready");
```



Animal first calls its own superclass: `Object()`. Every chain ends at Object.

Step 4 / 11

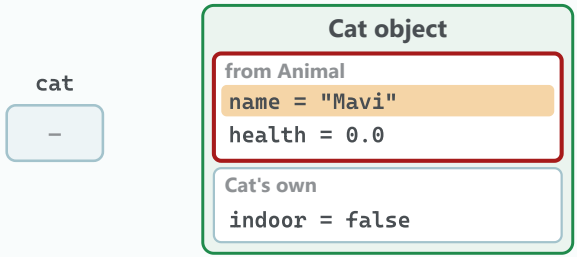
CALL STACK

- `Object()`
- `Animal("Mavi")`
- `Cat("Mavi", true)`
- `main()`

OUTPUT

Constructor Chain: `new Cat("Mavi", true)`

```
1  class Animal {
2      private String name;
3      private double health;
4      Animal(String name) {
5          super();   // calls Object()
6          this.name = name;
7          health = 100.0;
8          System.out.println("Animal
   constructor");
9      }
10 }
11 class Cat extends Animal {
12     private boolean indoor;
13     Cat(String name, boolean indoor) {
14         super(name);
15         this.indoor = indoor;
16         System.out.println("Cat constructor");
17     }
18 }
19 // in main
20 Cat cat = new Cat("Mavi", true);
21 System.out.println("Object ready");
```



Object() returns. Animal fills its own field: `name = "Mavi"`.

Step 5 / 11

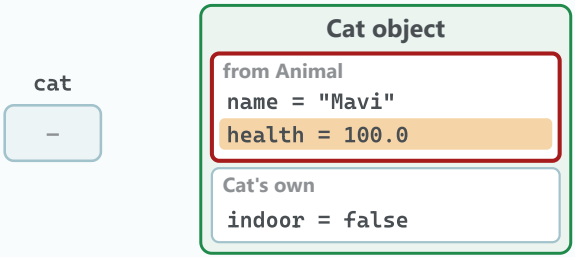
CALL STACK

- `Animal("Mavi")`
- `Cat("Mavi", true)`
- `main()`

OUTPUT

Constructor Chain: `new Cat("Mavi", true)`

```
1  class Animal {
2      private String name;
3      private double health;
4      Animal(String name) {
5          super();   // calls Object()
6          this.name = name;
▶          health = 100.0;
8          System.out.println("Animal
   constructor");
9      }
10 }
11 class Cat extends Animal {
12     private boolean indoor;
13     Cat(String name, boolean indoor) {
14         super(name);
15         this.indoor = indoor;
16         System.out.println("Cat constructor");
17     }
18 }
19 // in main
20 Cat cat = new Cat("Mavi", true);
21 System.out.println("Object ready");
```



`health = 100.0` .

Step 6 / 11

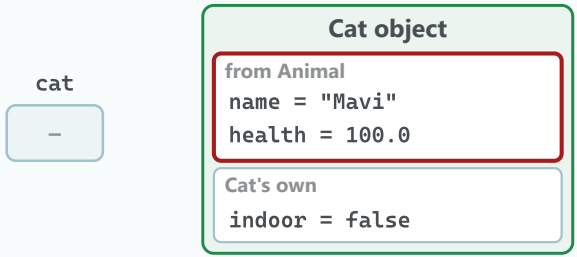
CALL STACK

- `Animal("Mavi")`
- `Cat("Mavi", true)`
- `main()`

OUTPUT

Constructor Chain: `new Cat("Mavi", true)`

```
1  class Animal {
2      private String name;
3      private double health;
4      Animal(String name) {
5          super();   // calls Object()
6          this.name = name;
7          health = 100.0;
▶          System.out.println("Animal
   constructor");
9      }
10 }
11 class Cat extends Animal {
12     private boolean indoor;
13     Cat(String name, boolean indoor) {
14         super(name);
15         this.indoor = indoor;
16         System.out.println("Cat constructor");
17     }
18 }
19 // in main
20 Cat cat = new Cat("Mavi", true);
21 System.out.println("Object ready");
```



The Animal constructor prints its message and returns: the **superclass printed first**.

Step 7 / 11

CALL STACK

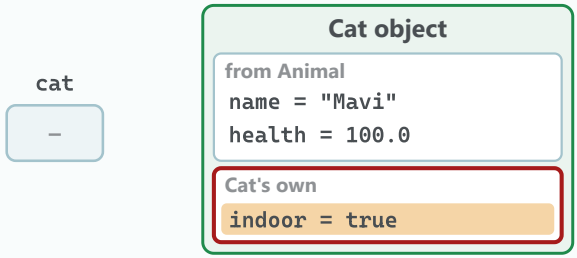
- `Animal("Mavi")`
- `Cat("Mavi", true)`
- `main()`

OUTPUT

```
Animal constructor
```

Constructor Chain: `new Cat("Mavi", true)`

```
1  class Animal {
2      private String name;
3      private double health;
4      Animal(String name) {
5          super();   // calls Object()
6          this.name = name;
7          health = 100.0;
8          System.out.println("Animal
   constructor");
9      }
10 }
11 class Cat extends Animal {
12     private boolean indoor;
13     Cat(String name, boolean indoor) {
14         super(name);
▶          this.indoor = indoor;
16         System.out.println("Cat constructor");
17     }
18 }
19 // in main
20 Cat cat = new Cat("Mavi", true);
21 System.out.println("Object ready");
```



The Cat constructor resumes: `indoor = true`.

Step 8 / 11

CALL STACK

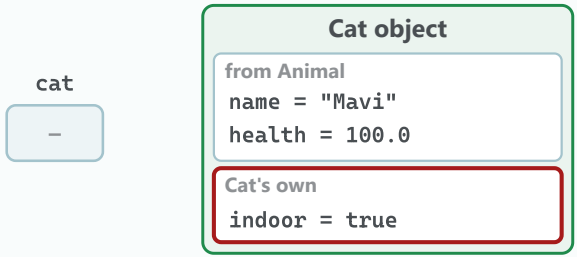
- `Cat("Mavi", true)`
- `main()`

OUTPUT

```
Animal constructor
```

Constructor Chain: `new Cat("Mavi", true)`

```
1  class Animal {
2      private String name;
3      private double health;
4      Animal(String name) {
5          super();   // calls Object()
6          this.name = name;
7          health = 100.0;
8          System.out.println("Animal
   constructor");
9      }
10 }
11 class Cat extends Animal {
12     private boolean indoor;
13     Cat(String name, boolean indoor) {
14         super(name);
15         this.indoor = indoor;
▶          System.out.println("Cat constructor");
17     }
18 }
19 // in main
20 Cat cat = new Cat("Mavi", true);
21 System.out.println("Object ready");
```



The Cat constructor prints its message.

Step 9 / 11

CALL STACK

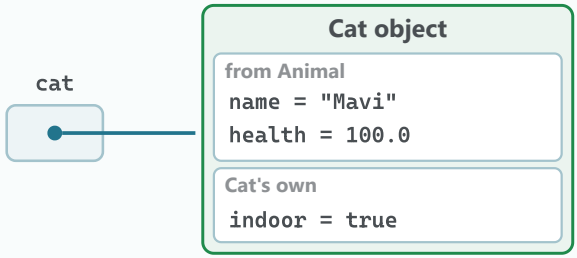
- `Cat("Mavi", true)`
- `main()`

OUTPUT

```
Animal constructor
Cat constructor
```

Constructor Chain: `new Cat("Mavi", true)`

```
1  class Animal {
2      private String name;
3      private double health;
4      Animal(String name) {
5          super();   // calls Object()
6          this.name = name;
7          health = 100.0;
8          System.out.println("Animal
   constructor");
9      }
10 }
11 class Cat extends Animal {
12     private boolean indoor;
13     Cat(String name, boolean indoor) {
14         super(name);
15         this.indoor = indoor;
16         System.out.println("Cat constructor");
17     }
18 }
19 // in main
▶  Cat cat = new Cat("Mavi", true);
21 System.out.println("Object ready");
```



The constructor is done; the object's address is stored in `cat`.

Step 10 / 11

CALL STACK

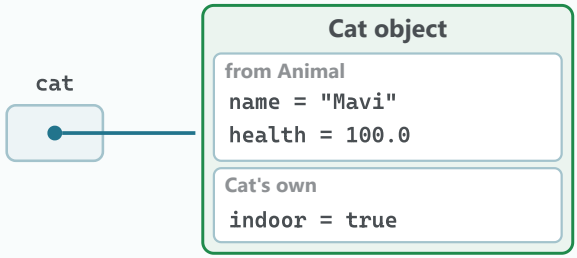
`main()`

OUTPUT

```
Animal constructor
Cat constructor
```

Constructor Chain: `new Cat("Mavi", true)`

```
1  class Animal {
2      private String name;
3      private double health;
4      Animal(String name) {
5          super();   // calls Object()
6          this.name = name;
7          health = 100.0;
8          System.out.println("Animal
   constructor");
9      }
10 }
11 class Cat extends Animal {
12     private boolean indoor;
13     Cat(String name, boolean indoor) {
14         super(name);
15         this.indoor = indoor;
16         System.out.println("Cat constructor");
17     }
18 }
19 // in main
20 Cat cat = new Cat("Mavi", true);
▶  System.out.println("Object ready");
```



Constructors finish **top-down**: Object → Animal → Cat.

Step 11 / 11

CALL STACK

```
main()
```

OUTPUT

```
Animal constructor
Cat constructor
Object ready
```

Implicit super() and Errors

1) THE SUPERCLASS HAS NO NO-ARG CONSTRUCTOR

```
class Animal {
    private String name;
    Animal(String name) { this.name = name; }
}
class Cat extends Animal {
    Cat(String name) {
        // the compiler inserts super(); here
        System.out.println("Cat");
    }
}
```

```
error: constructor Animal in class Animal
  cannot be applied to given types;
  required: String
  found:    no arguments
```

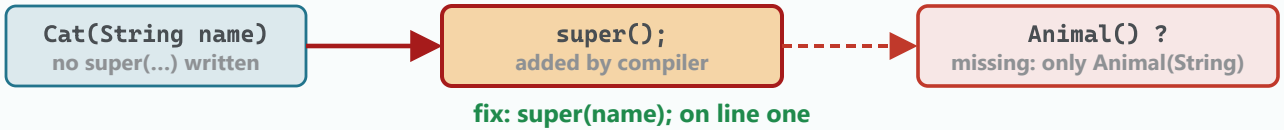
2) SUPER(...) IS NOT THE FIRST STATEMENT

```
Cat(String name) {
    System.out.println("Cat");
    super(name);
}
```

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

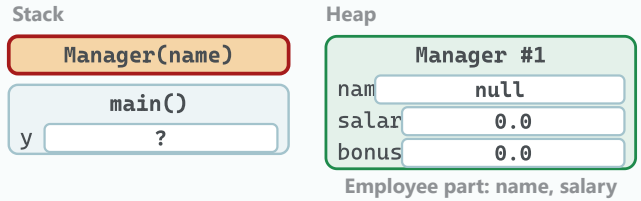
Fix: call the right superclass constructor explicitly on the first line of the subclass constructor: `super(name);`

For a class with no constructors, the compiler adds a no-arg default constructor; as soon as you write one, it does not.



`this(...)` and `super(...)`: Code Trace

```
 1 class Employee {
 2     private String name;  private double salary;
 3     Employee(String name) {
 4         this(name, 20000.0);
 5         System.out.println("Employee(String)");
 6     }
 7     Employee(String name, double salary) {
 8         this.name = name;  this.salary = salary;
 9         System.out.println("Employee(String, double)");
10     }
11     double getSalary() { return salary; }
12 }
13 class Manager extends Employee {
14     private double bonus;
15     Manager(String name) {
16         this(name, 5000.0);
17         System.out.println("Manager(String)");
18     }
19     Manager(String name, double bonus) {
20         super(name);
21         this.bonus = bonus;
22         System.out.println("Manager(String, double)");
23     }
24     double income() { return getSalary() + bonus; }
25 }
▶ Manager y = new Manager("Ayşe");   // main
27 System.out.println(y.income());
```



`new`: object allocated, fields default

`new` first allocates the object; all three fields hold default values. Then `Manager(String)` is called.

Step 1 / 9

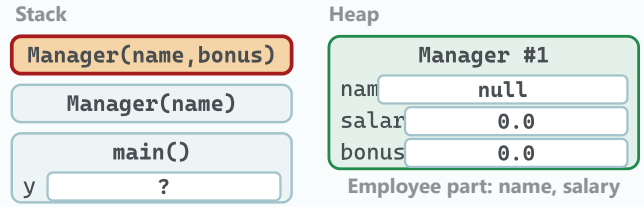
OUTPUT

VARIABLES

`y = ?`

`this(...)` and `super(...)`: Code Trace

```
1  class Employee {
2      private String name;  private double salary;
3      Employee(String name) {
4          this(name, 20000.0);
5          System.out.println("Employee(String)");
6      }
7      Employee(String name, double salary) {
8          this.name = name;  this.salary = salary;
9          System.out.println("Employee(String, double)");
10     }
11     double getSalary() { return salary; }
12 }
13 class Manager extends Employee {
14     private double bonus;
15     Manager(String name) {
▶          this(name, 5000.0);
17         System.out.println("Manager(String)");
18     }
19     Manager(String name, double bonus) {
20         super(name);
21         this.bonus = bonus;
22         System.out.println("Manager(String, double)");
23     }
24     double income() { return getSalary() + bonus; }
25 }
26 Manager y = new Manager("Ayşe");   // main
27 System.out.println(y.income());
```



`this( … )` → another constructor of the same class

The first line, `this(name, 5000.0)`, passes control to the two-argument constructor of the same class. `Manager(String)` waits before running its body.

Step 2 / 9

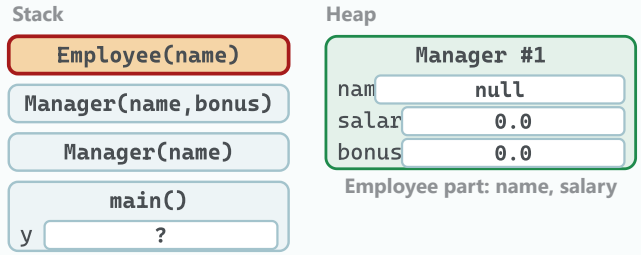
OUTPUT

VARIABLES

`y = ?`

`this(...)` and `super(...)`: Code Trace

```
1  class Employee {
2      private String name;  private double salary;
3      Employee(String name) {
4          this(name, 20000.0);
5          System.out.println("Employee(String)");
6      }
7      Employee(String name, double salary) {
8          this.name = name;  this.salary = salary;
9          System.out.println("Employee(String, double)");
10     }
11     double getSalary() { return salary; }
12 }
13 class Manager extends Employee {
14     private double bonus;
15     Manager(String name) {
16         this(name, 5000.0);
17         System.out.println("Manager(String)");
18     }
19     Manager(String name, double bonus) {
▶        super(name);
21         this.bonus = bonus;
22         System.out.println("Manager(String, double)");
23     }
24     double income() { return getSalary() + bonus; }
25 }
26 Manager y = new Manager("Ayşe");   // main
27 System.out.println(y.income());
```



`super(name)` → superclass constructor

`super(name)`: the one-argument constructor of the superclass `Employee`. Building the Manager part waits until the Employee part is done.

Step 3 / 9

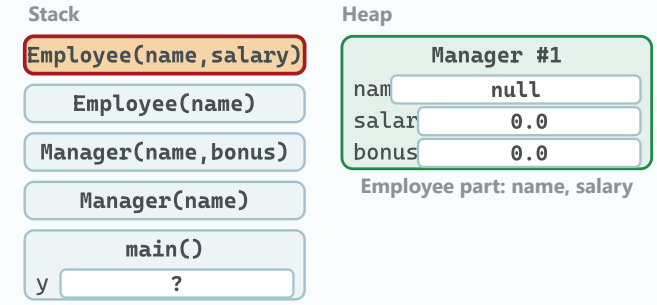
OUTPUT

VARIABLES

`y = ?`

`this(...)` and `super(...)`: Code Trace

```
1  class Employee {
2      private String name;  private double salary;
3      Employee(String name) {
4          this(name, 20000.0);
5          System.out.println("Employee(String)");
6      }
7      Employee(String name, double salary) {
8          this.name = name;  this.salary = salary;
9          System.out.println("Employee(String, double)");
10     }
11     double getSalary() { return salary; }
12 }
13 class Manager extends Employee {
14     private double bonus;
15     Manager(String name) {
16         this(name, 5000.0);
17         System.out.println("Manager(String)");
18     }
19     Manager(String name, double bonus) {
20         super(name);
21         this.bonus = bonus;
22         System.out.println("Manager(String, double)");
23     }
24     double income() { return getSalary() + bonus; }
25 }
26 Manager y = new Manager("Ayşe");   // main
27 System.out.println(y.income());
```



`this(...)` again; the chain ends in implicit `super()`

`Employee` also delegates with `this(...)`. That constructor's first line is neither `this` nor `super`, so the compiler inserts an implicit `super()`: `Object()` runs and returns.

Step 4 / 9

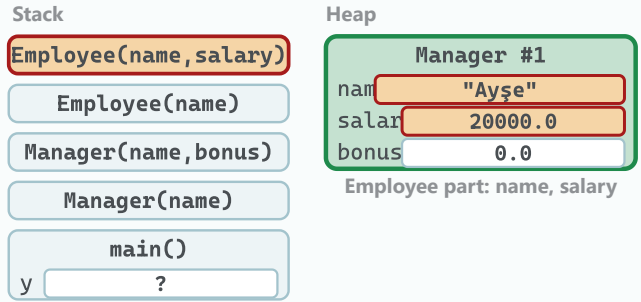
OUTPUT

VARIABLES

`y = ?`

`this(...)` and `super(...)`: Code Trace

```
1  class Employee {
2      private String name;  private double salary;
3      Employee(String name) {
4          this(name, 20000.0);
5          System.out.println("Employee(String)");
6      }
7      Employee(String name, double salary) {
▶          this.name = name;  this.salary = salary;
▶          System.out.println("Employee(String, double)");
10     }
11     double getSalary() { return salary; }
12 }
13 class Manager extends Employee {
14     private double bonus;
15     Manager(String name) {
16         this(name, 5000.0);
17         System.out.println("Manager(String)");
18     }
19     Manager(String name, double bonus) {
20         super(name);
21         this.bonus = bonus;
22         System.out.println("Manager(String, double)");
23     }
24     double income() { return getSalary() + bonus; }
25 }
26 Manager y = new Manager("Ayşe");   // main
27 System.out.println(y.income());
```



Employee part built

The first body runs here: `name` and `salary` are set and the first line is printed. The calls went bottom-up; the bodies complete starting from the topmost class.

Step 5 / 9

OUTPUT

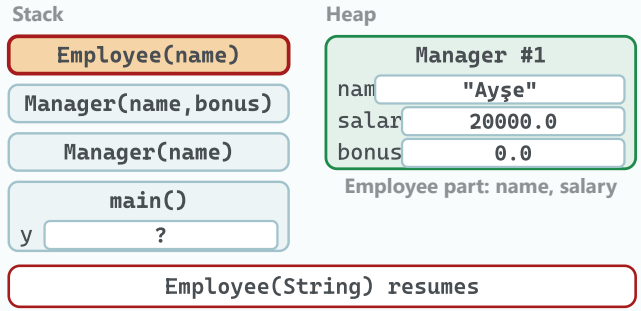
```
Employee(String, double)
```

VARIABLES

`y = ?`

`this(...)` and `super(...)`: Code Trace

```
 1 class Employee {
 2     private String name;  private double salary;
 3     Employee(String name) {
 4         this(name, 20000.0);
 5         System.out.println("Employee(String)");
 6     }
 7     Employee(String name, double salary) {
 8         this.name = name;  this.salary = salary;
 9         System.out.println("Employee(String, double)");
10     }
11     double getSalary() { return salary; }
12 }
13 class Manager extends Employee {
14     private double bonus;
15     Manager(String name) {
16         this(name, 5000.0);
17         System.out.println("Manager(String)");
18     }
19     Manager(String name, double bonus) {
20         super(name);
21         this.bonus = bonus;
22         System.out.println("Manager(String, double)");
23     }
24     double income() { return getSalary() + bonus; }
25 }
26 Manager y = new Manager("Ayşe");   // main
27 System.out.println(y.income());
```



The two-argument constructor returned; `Employee(String)` prints its line.

Step 6 / 9

OUTPUT

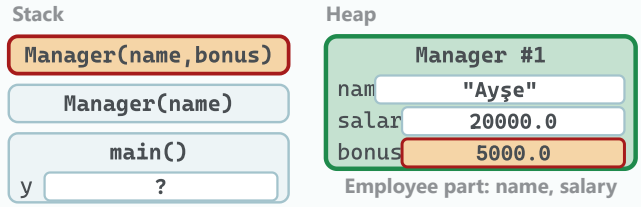
```
Employee(String, double)
Employee(String)
```

VARIABLES

`y = ?`

`this(...)` and `super(...)`: Code Trace

```
1  class Employee {
2      private String name;  private double salary;
3      Employee(String name) {
4          this(name, 20000.0);
5          System.out.println("Employee(String)");
6      }
7      Employee(String name, double salary) {
8          this.name = name;  this.salary = salary;
9          System.out.println("Employee(String, double)");
10     }
11     double getSalary() { return salary; }
12 }
13 class Manager extends Employee {
14     private double bonus;
15     Manager(String name) {
16         this(name, 5000.0);
17         System.out.println("Manager(String)");
18     }
19     Manager(String name, double bonus) {
20         super(name);
▶        this.bonus = bonus;
▶        System.out.println("Manager(String, double)");
23     }
24     double income() { return getSalary() + bonus; }
25 }
26 Manager y = new Manager("Ayşe");   // main
27 System.out.println(y.income());
```



building the Manager part

`super(name)` is done. Only now does the subclass set its own field: `bonus = 5000.0`.

Step 7 / 9

OUTPUT

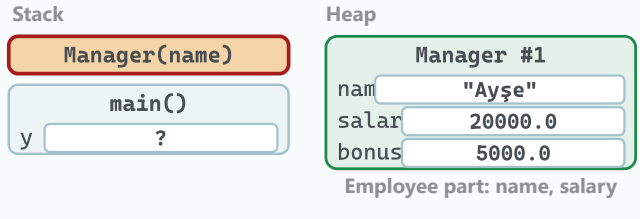
```
Employee(String, double)
Employee(String)
Manager(String, double)
```

VARIABLES

`y = ?`

`this(...)` and `super(...)`: Code Trace

```
1  class Employee {
2      private String name;  private double salary;
3      Employee(String name) {
4          this(name, 20000.0);
5          System.out.println("Employee(String)");
6      }
7      Employee(String name, double salary) {
8          this.name = name;  this.salary = salary;
9          System.out.println("Employee(String, double)");
10     }
11     double getSalary() { return salary; }
12 }
13 class Manager extends Employee {
14     private double bonus;
15     Manager(String name) {
16         this(name, 5000.0);
▶            System.out.println("Manager(String)");
18     }
19     Manager(String name, double bonus) {
20         super(name);
21         this.bonus = bonus;
22         System.out.println("Manager(String, double)");
23     }
24     double income() { return getSalary() + bonus; }
25 }
26 Manager y = new Manager("Ayşe");   // main
27 System.out.println(y.income());
```



`first constructor called finishes last`

Finally the first one called, `Manager(String)`, finishes. Four constructors ran, but only one object was built.

Step 8 / 9

OUTPUT

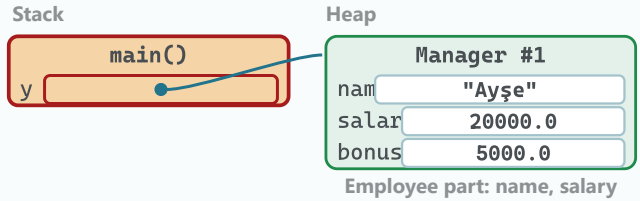
```
Employee(String, double)
Employee(String)
Manager(String, double)
Manager(String)
```

VARIABLES

`y = ?`

`this(...)` and `super(...)`: Code Trace

```
1  class Employee {
2      private String name;  private double salary;
3      Employee(String name) {
4          this(name, 20000.0);
5          System.out.println("Employee(String)");
6      }
7      Employee(String name, double salary) {
8          this.name = name;  this.salary = salary;
9          System.out.println("Employee(String, double)");
10     }
11     double getSalary() { return salary; }
12 }
13 class Manager extends Employee {
14     private double bonus;
15     Manager(String name) {
16         this(name, 5000.0);
17         System.out.println("Manager(String)");
18     }
19     Manager(String name, double bonus) {
20         super(name);
21         this.bonus = bonus;
22         System.out.println("Manager(String, double)");
23     }
24     double income() { return getSalary() + bonus; }
25 }
26 Manager y = new Manager("Ayşe");   // main
▶  System.out.println(y.income());
```



`income() = 20000.0 + 5000.0`

The object's address is stored in `y`. `income()` adds the inherited `getSalary()` to its own `bonus`: `25000.0`.

Step 9 / 9

OUTPUT

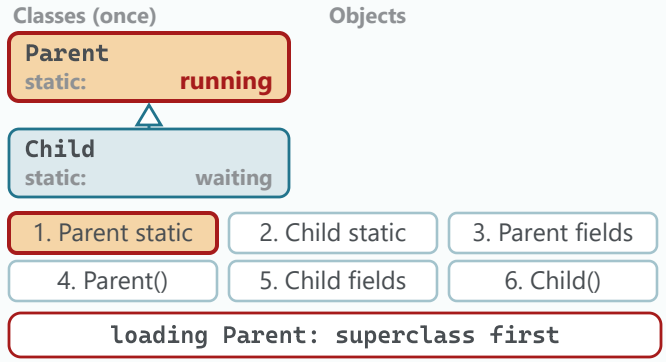
```
Employee(String, double)
Employee(String)
Manager(String, double)
Manager(String)
25000.0
```

VARIABLES

`y = →Manager #1`

Initialization Order with Inheritance: Code Trace

```
1  class Parent {
▶      static { System.out.println("Parent static"); }
3      int a = value("Parent field");
4      Parent() { System.out.print("Parent(), "); }
5      static int value(String s) {
6          System.out.print(s + ", ");
7          return 1;
8      }
9  }
10 class Child extends Parent {
11     static { System.out.println("Child static"); }
12     int b = value("Child field");
13     Child() { System.out.println("Child()"); }
14 }
15 // in main:
16 new Child();
17 new Child();
```



For the first `new Child()`, the JVM must initialize class `Child`. The superclass goes first: `Parent`'s `static` block runs.

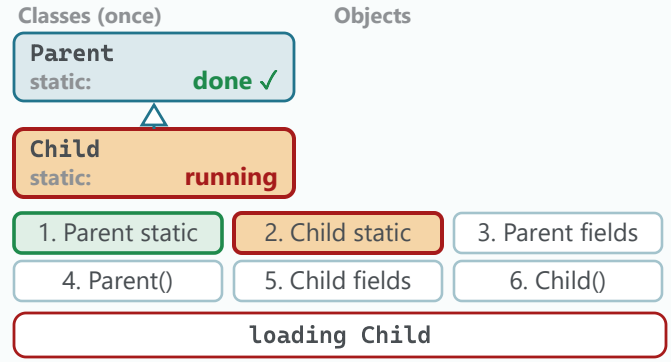
Step 1 / 7

OUTPUT

```
Parent static
```

Initialization Order with Inheritance: Code Trace

```
1  class Parent {
2      static { System.out.println("Parent
   static"); }
3      int a = value("Parent field");
4      Parent() { System.out.print("Parent(), "); }
5      static int value(String s) {
6          System.out.print(s + ", ");
7          return 1;
8      }
9  }
10 class Child extends Parent {
▶      static { System.out.println("Child static");
   }
12     int b = value("Child field");
13     Child() { System.out.println("Child()"); }
14 }
15 // in main:
16 new Child();
17 new Child();
```



Then `Child`'s `static` block. Class initialization happens **once** per program run.

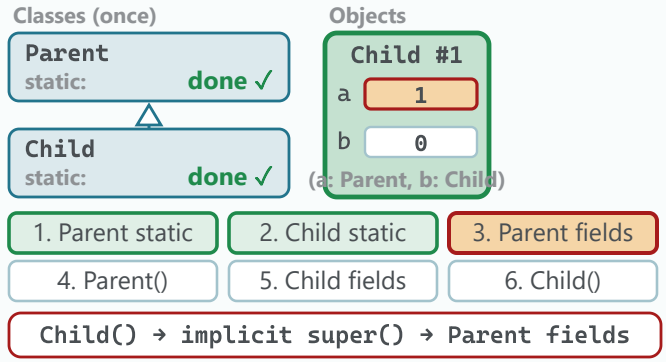
Step 2 / 7

OUTPUT

```
Parent static
Child static
```

Initialization Order with Inheritance: Code Trace

```
1  class Parent {
2      static { System.out.println("Parent
   static"); }
▶      int a = value("Parent field");
4      Parent() { System.out.print("Parent(), "); }
5      static int value(String s) {
▶          System.out.print(s + ", ");
7          return 1;
8      }
9  }
10 class Child extends Parent {
11     static { System.out.println("Child static");
   }
12     int b = value("Child field");
13     Child() { System.out.println("Child()"); }
14 }
15 // in main:
16 new Child();
17 new Child();
```



The object is allocated (`a` and `b` are 0). `Child()` first calls an implicit `super()`; Parent's field initializers run **before** the `Parent()` body: `a = value(...)`.

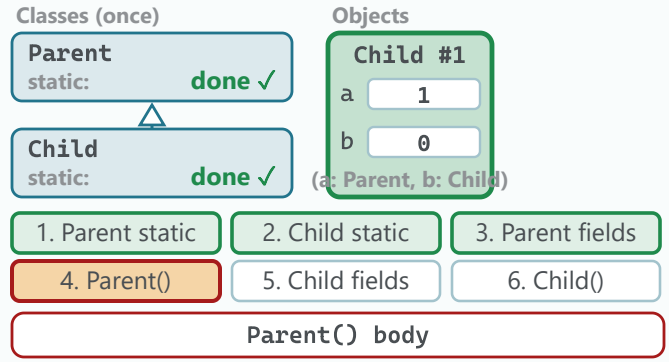
Step 3 / 7

OUTPUT

```
Parent static
Child static
Parent field,
```

Initialization Order with Inheritance: Code Trace

```
1  class Parent {
2      static { System.out.println("Parent
   static"); }
3      int a = value("Parent field");
▶      Parent() { System.out.print("Parent(), "); }
5      static int value(String s) {
6          System.out.print(s + ", ");
7          return 1;
8      }
9  }
10 class Child extends Parent {
11     static { System.out.println("Child static");
   }
12     int b = value("Child field");
13     Child() { System.out.println("Child()"); }
14 }
15 // in main:
16 new Child();
17 new Child();
```



Then the `Parent()` body. At this point `b` is still 0: the Child part has not been initialized at all.

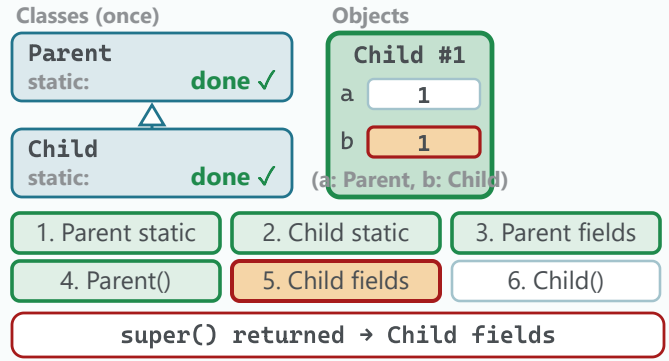
Step 4 / 7

OUTPUT

```
Parent static
Child static
Parent field, Parent(),
```

Initialization Order with Inheritance: Code Trace

```
1  class Parent {
2      static { System.out.println("Parent
   static"); }
3      int a = value("Parent field");
4      Parent() { System.out.print("Parent(), "); }
5      static int value(String s) {
6          System.out.print(s + ", ");
7          return 1;
8      }
9  }
10 class Child extends Parent {
11     static { System.out.println("Child static");
   }
▶      int b = value("Child field");
13     Child() { System.out.println("Child()"); }
14 }
15 // in main:
16 new Child();
17 new Child();
```



After `super()` returns, the subclass field initializers run: `b = value("Child field")`.

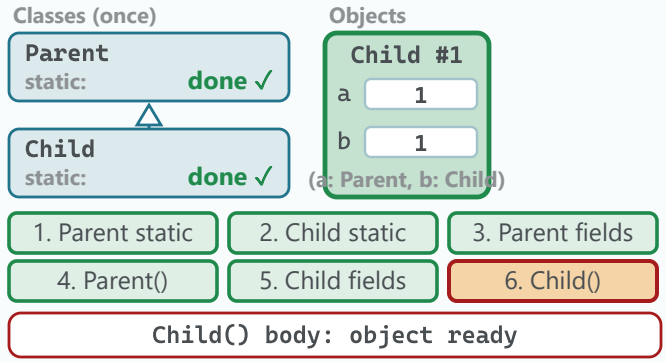
Step 5 / 7

OUTPUT

```
Parent static
Child static
Parent field, Parent(), Child field,
```

Initialization Order with Inheritance: Code Trace

```
1  class Parent {
2      static { System.out.println("Parent
   static"); }
3      int a = value("Parent field");
4      Parent() { System.out.print("Parent(), "); }
5      static int value(String s) {
6          System.out.print(s + ", ");
7          return 1;
8      }
9  }
10 class Child extends Parent {
11     static { System.out.println("Child static");
   }
12     int b = value("Child field");
▶      Child() { System.out.println("Child()"); }
14 }
15 // in main:
16 new Child();
17 new Child();
```



Last comes the `Child()` body. Order: superclass fields → superclass constructor → subclass fields → subclass constructor.

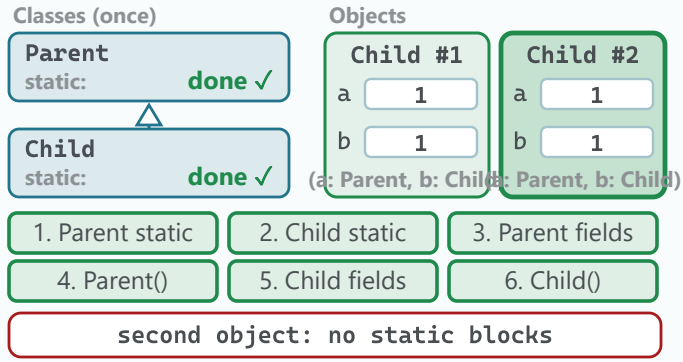
Step 6 / 7

OUTPUT

```
Parent static
Child static
Parent field, Parent(), Child field, Child()
```

Initialization Order with Inheritance: Code Trace

```
1  class Parent {
2      static { System.out.println("Parent
   static"); }
3      int a = value("Parent field");
4      Parent() { System.out.print("Parent(), "); }
5      static int value(String s) {
6          System.out.print(s + ", ");
7          return 1;
8      }
9  }
10 class Child extends Parent {
11     static { System.out.println("Child static");
   }
12     int b = value("Child field");
13     Child() { System.out.println("Child()"); }
14 }
15 // in main:
16 new Child();
▶  new Child();
```



Second `new Child()`: the classes are already initialized, so the `static` blocks **do not run again**. The four per-object steps repeat in the same order.

Step 7 / 7

OUTPUT

```
Parent static
Child static
Parent field, Parent(), Child field, Child()
Parent field, Parent(), Child field, Child()
```

Overriding a Method

Overriding: a subclass rewrites an inherited method **with the same signature** to give it its own behavior.

- Name and parameter list must be **identical**.
- The return type must be the same (or a subtype).
- Access **cannot be narrowed**: a `public` method stays `public`.
- `final`, `static` and `private` methods cannot be overridden.

```
class Shape {
    public void draw() {
        System.out.println("Drawing a shape.");
    }
}
class Rectangle extends Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a rectangle.");
    }
}
// in main
new Shape().draw();
new Rectangle().draw();
```

```
Drawing a shape.
Drawing a rectangle.
```

```
// in subclass: void draw()
error: draw() in Rectangle cannot override
  draw() in Shape
  attempting to assign weaker access
  privileges; was public
```

```
// in subclass: public int draw()
error: draw() in Rectangle cannot override
  draw() in Shape
  return type int is not compatible
  with void
```

Overriding vs. Overloading

	Overriding	Overloading
Where?	In a subclass, for an inherited method	In the same class (or a subclass)
Name	Same	Same
Parameters	**Same**	**Different** (number or type)
Return type	Same or a subtype	Any
Purpose	**Change** inherited behavior	Do the same job with different inputs
Resolved at	Run time (actual type of the object)	Compile time (argument types)

```
class Rectangle extends Shape {
    @Override
    public void draw() { ... }              // overriding
    public void draw(String color) { ... }  // overloading
}
```

OUTPUT OF `d.draw("Mavi")`

```
Drawing a blue rectangle.
```

Both ideas underlie **polymorphism**.

The @Override Annotation

- `@Override` is an **annotation**: it states that the method overrides an inherited method.
- It tells the compiler to **check** this: if the superclass has no such method, compilation fails.
- It makes the code easier to read; other developers see at once that the method comes from the superclass.
- It is optional, but you should **always write it**: it catches typos immediately.

```
public class Rectangle extends Shape {
    // override the inherited method
    @Override
    public void draw() {
        System.out.println("Drawing a rectangle.");
    }
}
```

TYPO: `Draw` (CAPITAL D)

```
    @Override
    public void Draw() { ... }
```

```
error: method does not override or implement
  a method from a supertype
```

Without `@Override`, `Draw()` would silently become a new method, and calls to `draw()` would still run Shape's version.

Overriding Rules in Detail

Aspect	Allowed in the subclass	`javac` if broken
Name and parameters	Exactly the same	No error: a new method (overload)
Return type	Same or a subtype (covariant)	`return type String is not compatible with Animal`
Access	Same or wider: `protected` → `public`	`attempting to assign weaker access privileges`
Checked exceptions	Same, more specific, or none	`overridden method does not throw Exception`
`static`	Cannot switch between instance and `static`	`overriding method is static` / `overridden method is static`
`final` method	Cannot be overridden	`overridden method is final`
`private` method	Invisible; a same-named method is unrelated	No error (error with `@Override`)

The common reason: a subclass object must work wherever a superclass object is expected. The superclass's promise (what it returns, who can call it, which exceptions it may throw) can be kept or **widened**, never **narrowed**.

Covariant Return Types

```
class Animal {
    Animal reproduce() { return new Animal(); }
    public String toString() { return "Animal"; }
}
class Cat extends Animal {
    @Override
    Cat reproduce() { return new Cat(); }  // subtype
    void meow() { System.out.println("meow"); }
    public String toString() { return "Cat"; }
}
// in main:
Cat k = new Cat();
k.reproduce().meow();   // no cast needed
Animal h = k;
System.out.println(h.reproduce());
```

```
meow
Cat
```

- An overriding method may return a **subtype** of the superclass method's return type: a **covariant return type**.
- A caller holding a `Cat` reference gets a `Cat` directly; no `(Cat)` cast is needed.
- A caller holding an `Animal` reference loses nothing: a `Cat` is also an `Animal`.
- This works only for reference types; you cannot return `long` instead of `int`.

UNRELATED TYPE: `String reproduce()`

```
error: reproduce() in Cat cannot override
  reproduce() in Animal
  return type String is not compatible
with Animal
```

`private` Methods Are Not Overridden

```
class Parent {
    private void greet() { System.out.println("Parent.greet"); }
    void call() { greet(); }
}
class Child extends Parent {
    void greet() { System.out.println("Child.greet"); } // new
}
// in main:
Child a = new Child();
a.call();
a.greet();
```

```
Parent.greet
Child.greet
```

IF `@Override` IS ADDED TO THE METHOD IN `Child`

```
error: method does not override or implement
  a method from a supertype
```

- A `private` method is invisible to subclasses; a same-named subclass method is an **unrelated** new one.
- In `Parent.call()`, `greet()` is bound at compile time to `Parent`'s `private` method, so `Parent.greet` is printed even for a `Child`.
- Using `@Override` catches this at compile time.

Likewise for `static` methods: a subclass `static` method with the same signature does not override the parent's, it only **hides** it.

Example: A Product Hierarchy

```
class Product {
    private String name;
    private double price;
    Product(String name, double price) {
        this.name = name; this.price = price;
    }
    double taxRate() { return 0.20; }
    double salePrice() { return price * (1 + taxRate()); }
    @Override public String toString() {
        double f = Math.round(salePrice() * 100) / 100.0;
        return name + ": " + f + " TL";
    }
}
class Food extends Product {
    Food(String name, double price) { super(name, price); }
    @Override double taxRate() { return 0.01; }
}
class SaleProduct extends Product {
    private double discount;
    SaleProduct(String name, double price, double discount) {
        super(name, price);
        this.discount = discount;
    }
    @Override double salePrice() {
        return super.salePrice() * (1 - discount);
    }
    @Override public String toString() {
        return super.toString()
            + " (" + (int) (discount * 100) + "% off)";
    }
}
```

```
Product[] cart = {new Product("Pen", 50),
    new Food("Bread", 10),
    new SaleProduct("Headphones", 1000, 0.25)};
for (Product u : cart) System.out.println(u);
```

```
Pen: 60.0 TL
Bread: 10.1 TL
Headphones: 900.0 TL (25% off)
```

- `Food` changes only `taxRate()`; `salePrice()` and `toString()` are not written there, yet they use 1%, because the call inside `Product` is chosen by the object's class.
- `SaleProduct` does not **replace** the superclass computation, it **extends** it: it applies the discount to `super.salePrice()`. 1000 × 1.20 × 0.75 = 900.

The `protected` Access Level

Modifier	Same class	Same package	Subclass (other package)	Everywhere
`private`	✓	✗	✗	✗
(none: package)	✓	✓	✗	✗
`protected`	✓	✓	✓	✗
`public`	✓	✓	✓	✓

- `public`: the members are accessible from everywhere. `private`: only from inside the class.
- `protected` sits between the two: the class itself, its **subclasses** and other classes **in the same package** can access it.
- A subclass can reach the `private` fields of its superclass only through inherited `public` or `protected` methods.

Good practice: keep fields `private`; expose the operations subclasses need as `protected` methods.

protected: Code Example

```java
package animals;

public class Animal {
    protected String name;  // subclasses
    private double health;  // Animal only

    public Animal(String name) {
        this.name = name;
        health = 100.0;
    }
    public double getHealth() { return health; }
    protected void heal(double amount) {
        health += amount;
    }
}
```

```java
package indoor;     // another package
import animals.Animal;

public class Cat extends Animal {
    public Cat(String name) { super(name); }
    public void sleep() {
        // protected field: allowed
        System.out.println(name + " sleeps.");
        // health += 5;  // private: error
        heal(5);   // protected method
        System.out.println("Health: "
                           + getHealth());
    }
}
```

OUTPUT OF `new Cat("Mavi").sleep();`

```
Mavi sleeps.
Health: 105.0
```

IF THE COMMENT IS REMOVED

```
error: health has private access in Animal
```

A Subtle `protected` Rule

```java
package animals;
public class Animal {
    protected String name;
    public Animal(String name) {
        this.name = name;
    }
}
```

```java
package indoor;
import animals.Animal;
public class Cat extends Animal {
    public Cat(String name) { super(name); }
    void compare(Cat k, Animal h) {
        System.out.println(name);    // own field
        System.out.println(k.name);  // via Cat
        System.out.println(h.name);  // via Animal
    }
}
```

```
indoor/Cat.java:8: error: name has protected access in Animal
        System.out.println(h.name);
```

- A subclass in another package can reach a `protected` member only through a reference of **its own type** (or its subtypes): `name` and `k.name` are fine.
- `h`, however, may point to any `Animal`, a `Dog` for example. `Cat` is not allowed to touch the internals of another subclass.
- Classes in the same package have no such restriction: there, `protected` includes package access.

Restricting Inheritance with `final`

- The `final` keyword can be used to **restrict** inheritance.
- A `final` **method** cannot be overridden in any subclass.
- No class can extend a `final` **class**. Examples: `String`, `Math`, `Integer`.

```
class Animal {
    public final void breathe() { ... }
}
class Cat extends Animal {
    public void breathe() { ... }
}
```

```
error: breathe() in Cat cannot override
  breathe() in Animal
  overridden method is final
```

```
final class Fixed { ... }
class Test extends Fixed { }
```

```
error: cannot inherit from final Fixed
```

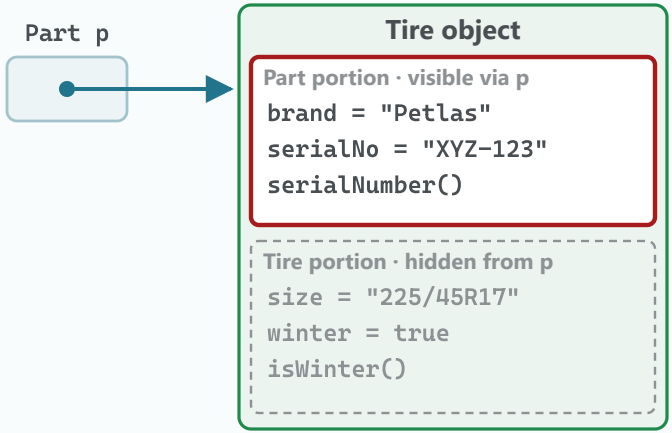
Why? Security and consistency: `String` must stay immutable, and no subclass should be able to break that promise.

On variables, `final` means something else: the value is assigned once (a constant).



Restricting an Object's View

- An object can be assigned to a variable of a **superclass type** (upcasting): every Tire is a Part.
- Only the methods **of the superclass** are then exposed.
- Whatever the object's real class, the variable's type decides what we **can call**.




```
Part p = new Tire("Petlas", "XYZ-123", "225/45R17", true);
System.out.println(p.serialNumber());   // allowed → XYZ-123
// p.isWinter()                         // not allowed
```

```
error: cannot find symbol
  symbol:   method isWinter()
  location: variable p of type Part
```

To reach a Tire-only method you need a type check (`instanceof`) followed by a downcast.

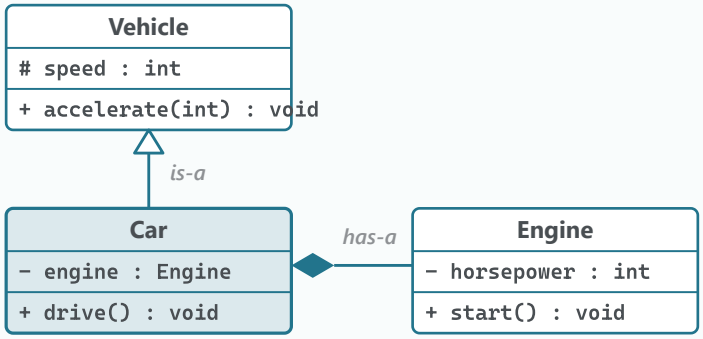
is-a and has-a Relationships

Is-a (inheritance)

- One class is derived from another; the subclass is **a kind of** the superclass.
- A subclass object can also be used as a superclass object.

Has-a (composition)

- A class has a field whose type is another class.
- The object holds **references** to other objects.




```java
// is-a relationship (inheritance)
public class Car extends Vehicle {
    // has-a relationship (composition)
    private Engine engine;

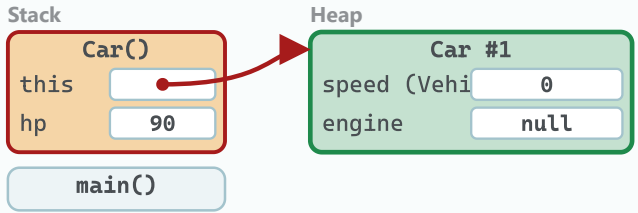
    public Car(int horsepower) {
        engine = new Engine(horsepower);
    }
    public void drive() {
        engine.start();   // delegate
        accelerate(50);          // inherited
        System.out.println("Speed: " + speed
                           + " km/h");
    }
}
// Engine: field horsepower and start():
//   "Engine started: " + horsepower + " hp"
```

```java
new Car(110).drive();
```

```
Engine started: 110 hp
Speed: 50 km/h
```

is-a and has-a: Code Trace

```
1  class Engine {
2      private int hp;
3      Engine(int hp) { this.hp = hp; }
4      void start() {
5          System.out.println("Engine: " + hp + " hp");
6      }
7  }
8  class Car extends Vehicle {        // is-a
9      private Engine engine;           // has-a
10     Car(int hp) { engine = new Engine(hp); }
11     void drive() {
12         engine.start();  accelerate(50);
13         System.out.println("Speed: " + speed);
14     }
15 }
▶  new Car(90).drive();   // main
```



Vehicle part (speed) + Car part (engine)

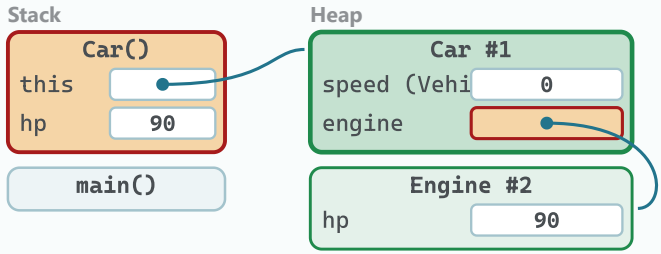
`new Car(90)` : one object, holding `speed` from the superclass `Vehicle` and its own `engine` field (**is-a**).

Step 1 / 6

OUTPUT

is-a and has-a: Code Trace

```
 1 class Engine {
 2     private int hp;
 3     Engine(int hp) { this.hp = hp; }
 4     void start() {
 5         System.out.println("Engine: " + hp + " hp");
 6     }
 7 }
 8 class Car extends Vehicle {        // is-a
 9     private Engine engine;           // has-a
10     Car(int hp) { engine = new Engine(hp); }
11     void drive() {
12         engine.start();  accelerate(50);
13         System.out.println("Speed: " + speed);
14     }
15 }
16 new Car(90).drive();   // main
```



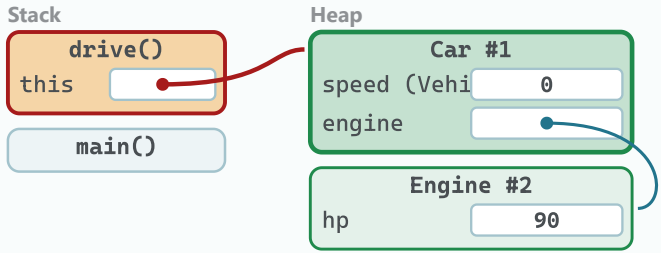
The constructor creates a second object: a car **is not** an engine, it **has** one (**has-a**). The link is a reference.

Step 2 / 6

OUTPUT

is-a and has-a: Code Trace

```
 1 class Engine {
 2     private int hp;
 3     Engine(int hp) { this.hp = hp; }
 4     void start() {
 5         System.out.println("Engine: " + hp + " hp");
 6     }
 7 }
 8 class Car extends Vehicle {        // is-a
 9     private Engine engine;           // has-a
10     Car(int hp) { engine = new Engine(hp); }
▶    void drive() {
12         engine.start();  accelerate(50);
13         System.out.println("Speed: " + speed);
14     }
15 }
▶ new Car(90).drive();   // main
```



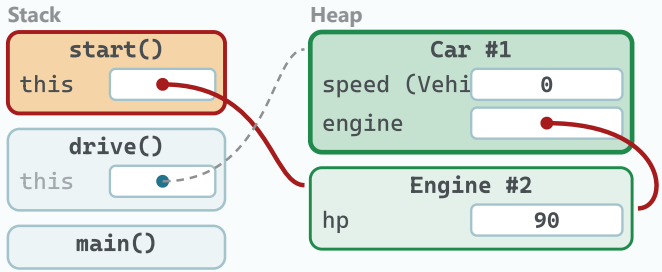
`drive()` runs on the Car object.

Step 3 / 6

OUTPUT

is-a and has-a: Code Trace

```
1  class Engine {
2      private int hp;
3      Engine(int hp) { this.hp = hp; }
▶      void start() {
▶          System.out.println("Engine: " + hp + " hp");
6      }
7  }
8  class Car extends Vehicle {        // is-a
9      private Engine engine;           // has-a
10     Car(int hp) { engine = new Engine(hp); }
11     void drive() {
▶          engine.start();  accelerate(50);
13         System.out.println("Speed: " + speed);
14     }
15 }
16 new Car(90).drive();   // main
```



`engine.start()` → the **Engine** object does the work

`engine.start()` : Car does not do the work itself; it follows the reference and **delegates** to the Engine object.

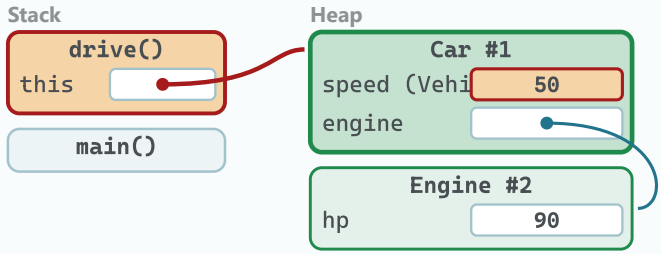
Step 4 / 6

OUTPUT

```
Engine: 90 hp
```

is-a and has-a: Code Trace

```
1  class Engine {
2      private int hp;
3      Engine(int hp) { this.hp = hp; }
4      void start() {
5          System.out.println("Engine: " + hp + " hp");
6      }
7  }
8  class Car extends Vehicle {        // is-a
9      private Engine engine;           // has-a
10     Car(int hp) { engine = new Engine(hp); }
11     void drive() {
▶          engine.start();  accelerate(50);
13         System.out.println("Speed: " + speed);
14     }
15 }
16 new Car(90).drive();   // main
```



`accelerate(50): method inherited from Vehicle`

`accelerate(50)` is not written in Car; it is inherited from `Vehicle` and sets the same object's `speed` to 50.

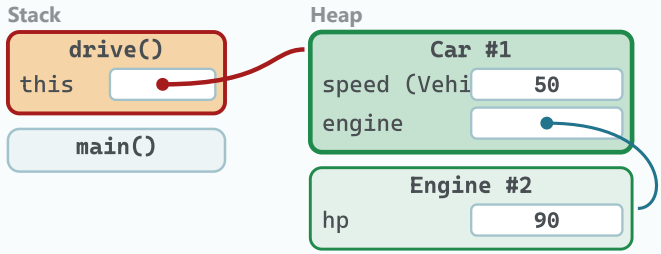
Step 5 / 6

OUTPUT

```
Engine: 90 hp
```

is-a and has-a: Code Trace

```
 1 class Engine {
 2     private int hp;
 3     Engine(int hp) { this.hp = hp; }
 4     void start() {
 5         System.out.println("Engine: " + hp + " hp");
 6     }
 7 }
 8 class Car extends Vehicle {        // is-a
 9     private Engine engine;           // has-a
10     Car(int hp) { engine = new Engine(hp); }
11     void drive() {
12         engine.start();  accelerate(50);
▶           System.out.println("Speed: " + speed);
14     }
15 }
16 new Car(90).drive();   // main
```



Inheritance (`speed` , `accelerate`) and composition (`engine`) worked together in one class.

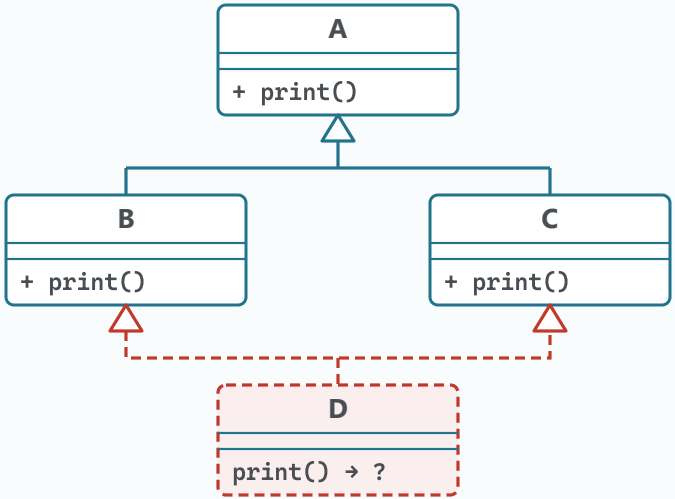
Step 6 / 6

OUTPUT

```
Engine: 90 hp
Speed: 50
```

Multiple Inheritance

- Java **does not support multiple inheritance** of classes: a class extends exactly **one** class.
- The **diamond problem**: if B and C override `print()` differently, which one does D get?
- When you need features of several classes: use **composition** and **interfaces** (a class can implement many interfaces).




```
class D extends B, C { }   // invalid
```

```
error: '{' expected
```

```
interface Chargeable { void charge(); }
class ElectricCar extends Car
                    implements Chargeable {
    @Override
    public void charge() {
        System.out.println("Charging.");
    }
}
```

Design Principles for Inheritance

The “is-a” test

If “every X is a Y” is false, do not use inheritance. A Car **is not** an Engine; it **has** one → composition.

Each class manages its own data

Superclass fields get their values through `super(...)` and superclass methods; fields stay `private`.

Override the general methods

Every meaningful class should override `toString` and `equals` (and, with it, `hashCode`).

Common concepts go into abstract classes

Concepts shared by subclasses should be modeled with **abstract** classes, if “Animal” on its own should never be an object.

Inheritance or Composition?

	Inheritance (`extends`)	**Composition (object as a field)**
Relationship	is-a: “every X is a Y”	has-a: “an X has a Y”
Coupling	Tight: the subclass depends on the superclass’s **implementation** too	Loose: depends only on the inner object’s `public` methods
Reuse	Automatic: every `public` method comes along (wanted or not)	Selective: you write only the methods you want to forward
Change	The superclass is fixed at compile time	The field can be swapped for another object at run time
Use in place of the supertype	Directly: `Animal h = new Cat()`	If both implement a common interface

Use inheritance

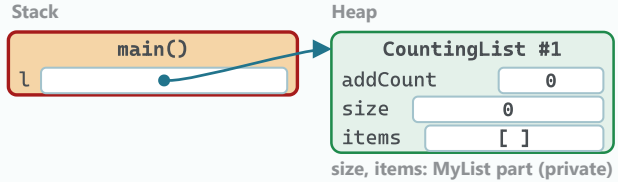
when the is-a relationship truly holds and the superclass was designed and documented for extension: `Cat` – `Animal`.

Use composition

when you only want to reuse another class’s code: a `Stack` **is not** an `ArrayList`; it keeps a list inside and offers only `add` / `remove`.

Fragile Base Class: Code Trace

```
 1 class MyList {
 2     private String[] items = new String[10];
 3     private int size;
 4     void add(String s) { items[size++] = s; }
 5     void addAll(String[] arr) {
 6         for (String s : arr) add(s);
 7     }
 8     int size() { return size; }
 9 }
10 class CountingList extends MyList {
11     int addCount = 0;
12     @Override void add(String s) {
13         addCount++;
14         super.add(s);
15     }
16     @Override void addAll(String[] arr) {
17         addCount += arr.length;
18         super.addAll(arr);
19     }
20 }
▶ CountingList l = new CountingList();   // main
22 l.addAll(new String[] {"a", "b", "c"});
23 System.out.println(l.addCount + " " + l.size());
```



`adding three items: expected count 3`

`CountingList` extends `MyList` to count the items added. Both add methods are overridden.

Step 1 / 7

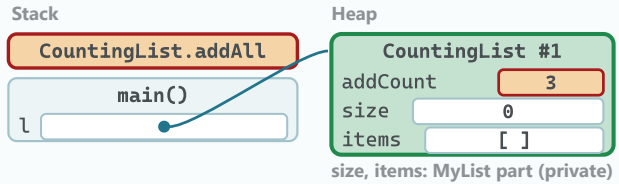
OUTPUT

VARIABLES

`addCount = 0` `size = 0`

Fragile Base Class: Code Trace

```
1  class MyList {
2      private String[] items = new String[10];
3      private int size;
4      void add(String s) { items[size++] = s; }
5      void addAll(String[] arr) {
6          for (String s : arr) add(s);
7      }
8      int size() { return size; }
9  }
10 class CountingList extends MyList {
11     int addCount = 0;
12     @Override void add(String s) {
13         addCount++;
14         super.add(s);
15     }
16     @Override void addAll(String[] arr) {
▶           addCount += arr.length;
18         super.addAll(arr);
19     }
20 }
21 CountingList l = new CountingList();   // main
22 l.addAll(new String[] {"a", "b", "c"});
23 System.out.println(l.addCount + " " + l.size());
```



`arr.length = 3 added`

The subclass's `addAll` adds the array length to the counter: 3. Then it hands the work to the superclass.

Step 2 / 7

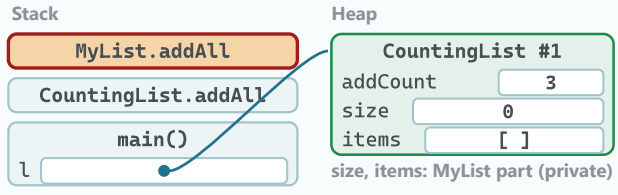
OUTPUT

VARIABLES

`addCount = 3` `size = 0`

Fragile Base Class: Code Trace

```
1  class MyList {
2      private String[] items = new String[10];
3      private int size;
4      void add(String s) { items[size++] = s; }
5      void addAll(String[] arr) {
▶          for (String s : arr) add(s);
7      }
8      int size() { return size; }
9  }
10 class CountingList extends MyList {
11     int addCount = 0;
12     @Override void add(String s) {
13         addCount++;
14         super.add(s);
15     }
16     @Override void addAll(String[] arr) {
17         addCount += arr.length;
▶          super.addAll(arr);
19     }
20 }
21 CountingList l = new CountingList();   // main
22 l.addAll(new String[] {"a", "b", "c"});
23 System.out.println(l.addCount + " " + l.size());
```



`MyList.addAll` calls `add(s)`, i.e. `this.add(s)`, for each item. That is an **implementation detail** of the superclass.

Step 3 / 7

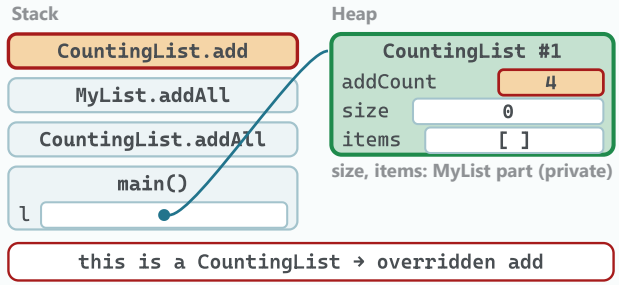
OUTPUT

VARIABLES

`addCount = 3` `size = 0` `s = "a"`

Fragile Base Class: Code Trace

```
 1 class MyList {
 2     private String[] items = new String[10];
 3     private int size;
 4     void add(String s) { items[size++] = s; }
 5     void addAll(String[] arr) {
 6         for (String s : arr) add(s);
 7     }
 8     int size() { return size; }
 9 }
10 class CountingList extends MyList {
11     int addCount = 0;
12     @Override void add(String s) {
▶           addCount++;
14         super.add(s);
15     }
16     @Override void addAll(String[] arr) {
17         addCount += arr.length;
18         super.addAll(arr);
19     }
20 }
21 CountingList l = new CountingList();   // main
22 l.addAll(new String[] {"a", "b", "c"});
23 System.out.println(l.addCount + " " + l.size());
```



The object is a `CountingList`, so the call goes to the subclass's `add`. The same item is counted **a second time**: 4.

Step 4 / 7

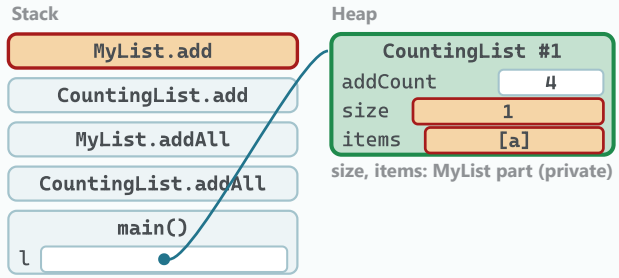
OUTPUT

VARIABLES

`addCount = 4` `size = 0` `s = "a"`

Fragile Base Class: Code Trace

```
 1 class MyList {
 2     private String[] items = new String[10];
 3     private int size;
 4     void add(String s) { items[size++] = s; }
 5     void addAll(String[] arr) {
 6         for (String s : arr) add(s);
 7     }
 8     int size() { return size; }
 9 }
10 class CountingList extends MyList {
11     int addCount = 0;
12     @Override void add(String s) {
13         addCount++;
14         super.add(s);
15     }
16     @Override void addAll(String[] arr) {
17         addCount += arr.length;
18         super.addAll(arr);
19     }
20 }
21 CountingList l = new CountingList();   // main
22 l.addAll(new String[] {"a", "b", "c"});
23 System.out.println(l.addCount + " " + l.size());
```



`super.add(s)` stores the item in the array: `size = 1`.

Step 5 / 7

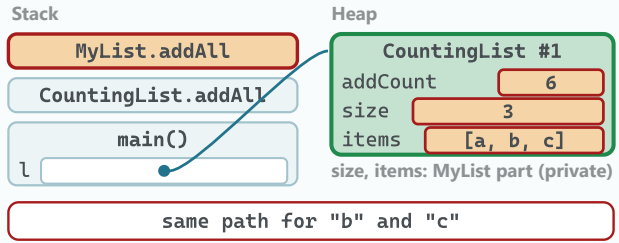
OUTPUT

VARIABLES

`addCount = 4` `size = 1` `s = "a"`

Fragile Base Class: Code Trace

```
1  class MyList {
2      private String[] items = new String[10];
3      private int size;
4      void add(String s) { items[size++] = s; }
5      void addAll(String[] arr) {
▶          for (String s : arr) add(s);
7      }
8      int size() { return size; }
9  }
10 class CountingList extends MyList {
11     int addCount = 0;
12     @Override void add(String s) {
13         addCount++;
14         super.add(s);
15     }
16     @Override void addAll(String[] arr) {
17         addCount += arr.length;
18         super.addAll(arr);
19     }
20 }
21 CountingList l = new CountingList();   // main
22 l.addAll(new String[] {"a", "b", "c"});
23 System.out.println(l.addCount + " " + l.size());
```



"b" and "c" take the same path: each is counted once more. When the loop ends the counter is 6 and size is 3.

Step 6 / 7

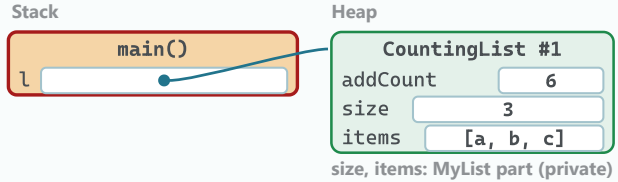
OUTPUT

VARIABLES

`addCount = 6` `size = 3`

Fragile Base Class: Code Trace

```
1  class MyList {
2      private String[] items = new String[10];
3      private int size;
4      void add(String s) { items[size++] = s; }
5      void addAll(String[] arr) {
6          for (String s : arr) add(s);
7      }
8      int size() { return size; }
9  }
10 class CountingList extends MyList {
11     int addCount = 0;
12     @Override void add(String s) {
13         addCount++;
14         super.add(s);
15     }
16     @Override void addAll(String[] arr) {
17         addCount += arr.length;
18         super.addAll(arr);
19     }
20 }
21 CountingList l = new CountingList();   // main
22 l.addAll(new String[] {"a", "b", "c"});
▶  System.out.println(l.addCount + " " + l.size());
```



`three items, count 6: fragile base class`

Wrong result: `6 3`. The subclass depended on how the superclass uses its own methods. If a new version of the superclass changed the loop, the result would change again: the **fragile base class** problem.

Step 7 / 7

OUTPUT

```
6 3
```

VARIABLES

`addCount = 6` `size = 3`

The Composition Fix: Forwarding

```
class CountingList2 {
    private final MyList list = new MyList(); // has-a
    private int addCount = 0;
    void add(String s) {
        addCount++;
        list.add(s);            // forwards
    }
    void addAll(String[] arr) {
        addCount += arr.length;
        list.addAll(arr);  // forwards
    }
    int getAddCount() { return addCount; }
    int size() { return list.size(); }
}
```

```
CountingList2 l2 = new CountingList2();
l2.addAll(new String[] {"a", "b", "c"});
System.out.println(l2.getAddCount()
    + " " + l2.size());
```

```
3 3
```

- The new class does not extend `MyList`; it **has** a `MyList`. Each method does its own work first, then **forwards** the call to the inner object.
- The `add` calls inside `list.addAll(...)` run on a plain `MyList` object; they never come back to `CountingList2`. The count is right: 3.
- Even if `MyList`'s internals change, the result does not: the dependency is only on `public` methods.
- The price: a `CountingList2` cannot be passed where a `MyList` is expected. If needed, both implement a common interface.

The Liskov Substitution Principle (LSP)

```
class Rectangle {
    protected int width, height;
    void setWidth(int w) { width = w; }
    void setHeight(int h) { height = h; }
    int area() { return width * height; }
}
class Square extends Rectangle {
    @Override void setWidth(int k) { width = height = k; }
    @Override void setHeight(int k) { width = height = k; }
}
static void resize(Rectangle d) {
    d.setWidth(5);
    d.setHeight(4);
    System.out.println("Expected 20, got "
        + d.area());
}
// in main:
resize(new Rectangle());
resize(new Square());
```

```
Expected 20, got 20
Expected 20, got 16
```

- **Liskov substitution principle**: code written against a superclass type must also work **correctly** with a subclass object. Compiling is not enough.
- In mathematics every square is a rectangle. But this `Rectangle` class promises that "width and height change independently"; `Square` cannot keep that promise.
- `resize` is not at fault; the fault lies in the subclass that breaks the promise.
- Fixes: do not derive `Square` from `Rectangle` (both extend `Shape`), or make the classes immutable.

Apply the is-a test to the class's **behavior**, not to the real world: "Can a Square answer every Rectangle method the way a Rectangle does?"

The Object Class

- Every class extends `java.lang.Object`, **indirectly** if no `extends` is written.
- So every object comes with the methods below **ready to use**.
- Their default versions are rarely useful; meaningful classes override them.

```
class Point { }
// what the compiler sees:
class Point extends Object { }
```

Method	Default behavior
`String toString()`	Class name + `@` + hashCode in hexadecimal
`boolean equals(Object o)`	Compares references: `this == o`
`int hashCode()`	An integer specific to the object (usually different for each object)
`Class getClass()`	The object's run-time class (final; cannot be overridden)

The `toString` Method

- `public String toString()` : returns the object as text. `println(obj)` and `"..." + obj` call it **automatically**.
- If not overridden, it returns a string with the class name and the object's hashCode.

NOT OVERRIDDEN

```
class Point {
    private int x, y;
    Point(int x, int y) {
        this.x = x; this.y = y;
    }
}
// in main
System.out.println(new Point(3, 4));
```

```
Point@28a418fc
```

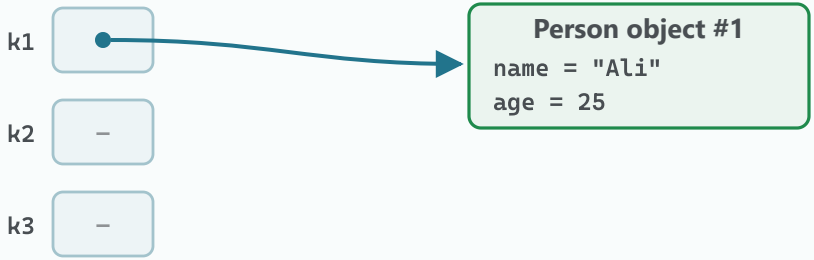
The value after @ may change from run to run and means nothing to a reader. Printing a `null` reference does not call `toString` ; it prints `null` .

OVERRIDDEN

```
class Point {
    private int x, y;
    Point(int x, int y) {
        this.x = x; this.y = y;
    }
    @Override
    public String toString() {
        return "Point[x=" + x + ", y=" + y + "]";
    }
}
// in main
Point n = new Point(3, 4);
System.out.println(n);
System.out.println("Position: " + n);
```

```
Point[x=3, y=4]
Position: Point[x=3, y=4]
```

== or equals?



`new` creates an object; `k1` holds its **address** (reference).

Step 1 / 6

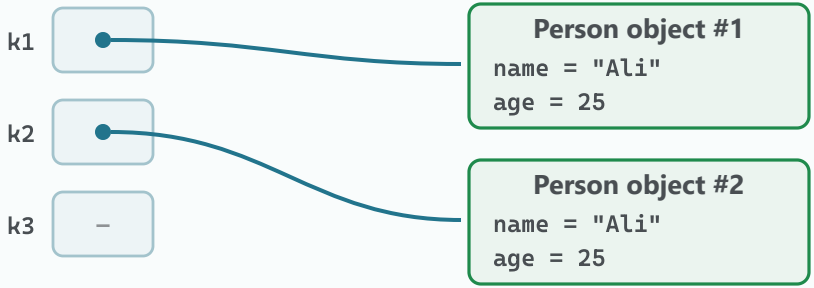
```
▶ Person k1 = new Person("Ali", 25);
2 Person k2 = new Person("Ali", 25);
3 Person k3 = k1;
4 System.out.println(k1 == k2);
5 System.out.println(k1 == k3);
6 System.out.println(k1.equals(k2));
```

VARIABLES

`k1 = →#1`

OUTPUT

== or equals?



Second `new` : same contents, but a **separate** object. `k2` points to it.

Step 2 / 6

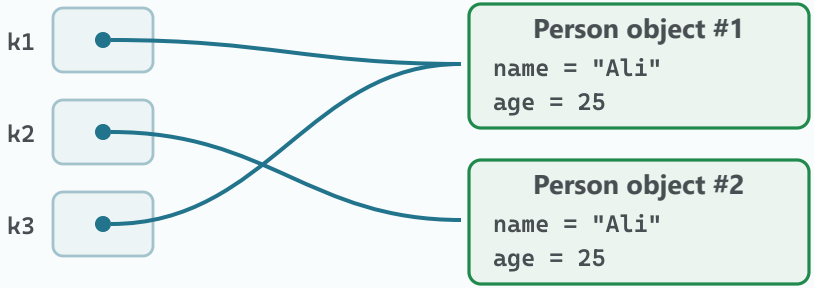
```
1 Person k1 = new Person("Ali", 25);
▶ Person k2 = new Person("Ali", 25);
3 Person k3 = k1;
4 System.out.println(k1 == k2);
5 System.out.println(k1 == k3);
6 System.out.println(k1.equals(k2));
```

VARIABLES

`k1 = →#1` `k2 = →#2`

OUTPUT

== or equals?



`k3 = k1` copies no object, only the address. k1 and k3 point to the **same** object.

Step 3 / 6

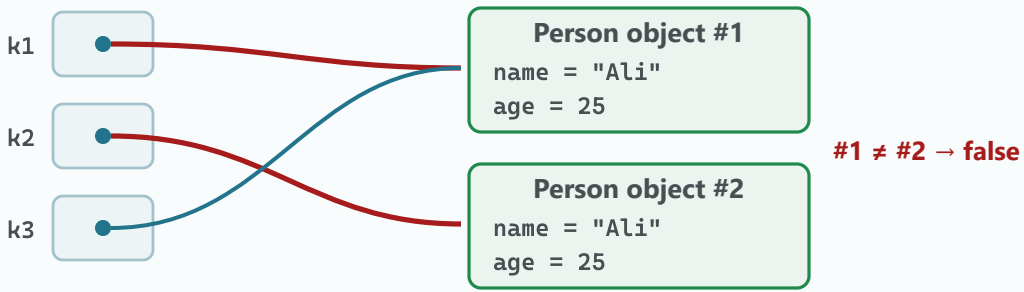
```
1 Person k1 = new Person("Ali", 25);
2 Person k2 = new Person("Ali", 25);
▶ Person k3 = k1;
4 System.out.println(k1 == k2);
5 System.out.println(k1 == k3);
6 System.out.println(k1.equals(k2));
```

VARIABLES

`k1 = →#1` `k2 = →#2` `k3 = →#1`

OUTPUT

== or equals?



`==` compares references: k1 and k2 point to different objects → `false`.

Step 4 / 6

```
1 Person k1 = new Person("Ali", 25);
2 Person k2 = new Person("Ali", 25);
3 Person k3 = k1;
▶ System.out.println(k1 == k2);
5 System.out.println(k1 == k3);
6 System.out.println(k1.equals(k2));
```

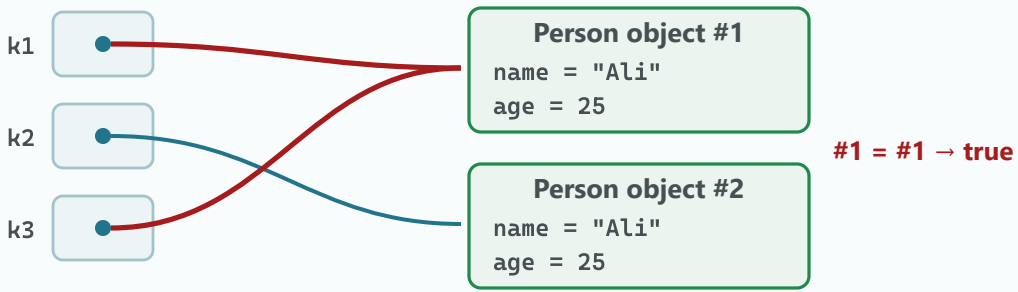
VARIABLES

k1 = →#1 | k2 = →#2 | k3 = →#1

OUTPUT

```
false
```

== or equals?



k1 and k3 point to the same object → `true`.

Step 5 / 6

```
1 Person k1 = new Person("Ali", 25);
2 Person k2 = new Person("Ali", 25);
3 Person k3 = k1;
4 System.out.println(k1 == k2);
▶ System.out.println(k1 == k3);
6 System.out.println(k1.equals(k2));
```

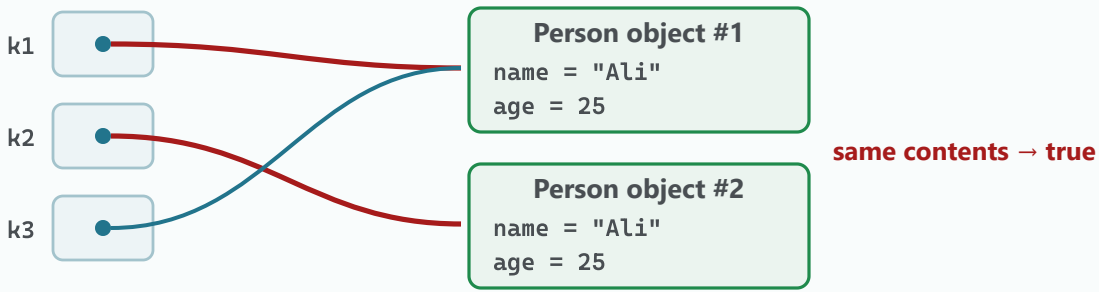
VARIABLES

k1 = →#1 | k2 = →#2 | k3 = →#1

OUTPUT

```
false
true
```

== or equals?



Since Person overrides `equals`, it compares the **contents** (name, age) → `true`. Without the override it would be `false`.

Step 6 / 6

```
1 Person k1 = new Person("Ali", 25);
2 Person k2 = new Person("Ali", 25);
3 Person k3 = k1;
4 System.out.println(k1 == k2);
5 System.out.println(k1 == k3);
▶ System.out.println(k1.equals(k2));
```

VARIABLES

`k1 = →#1` `k2 = →#2` `k3 = →#1`

OUTPUT

```
false
true
true
```

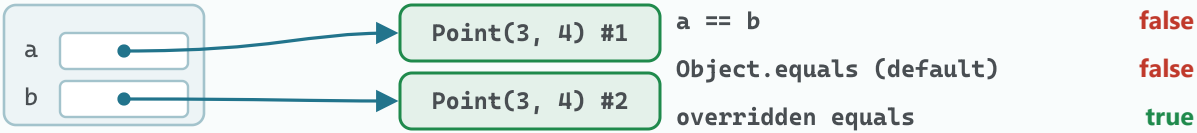
The `equals` Method

- `public boolean equals(Object other)`: tells whether two objects are **equivalent**.
- By default it compares only references: unless overridden it behaves like the `==` operator.
- Overriding changes this: the method should compare the two objects' **state** (their fields) and return `true` if they are equal.
- The parameter has type `Object`, so do a **type check** first to rule out bad cases.

THE EQUALS CONTRACT

Reflexive	`x.equals(x)` → true
Symmetric	`x.equals(y)` ⇔ `y.equals(x)`
Transitive	if x = y and y = z, then x = z
Consistent	The result stays the same while the fields do not change
null	`x.equals(null)` → false

The same rule applies to `String`: compare with `s1.equals(s2)`, not `s1 == s2`.



Writing `equals`

TYPE CHECK WITH `instanceof`

```
public boolean equals(Object obj) {
    if (obj instanceof Point) {
        // obj is a Point;
        // cast and compare
        Point other = (Point) obj;
        return x == other.x && y == other.y;
    } else {
        // obj is not a Point;
        // cannot be equal
        return false;
    }
}
```

- The parameter type must be `Object`; writing `equals(Point n)` is **overloading**, not overriding.
- The cast `(Point) obj` is safe only after the type check.

TEST

```
Point a = new Point(3, 4);
Point b = new Point(3, 4);
System.out.println(a == b);
System.out.println(a.equals(b));
System.out.println(a.equals("3,4"));
System.out.println(a.equals(null));
```

```
false
true
false
false
```

`"3,4"` is not a Point → `false`. Since `null instanceof Point` is also `false`, no separate null check is needed.

`equals` with `getClass()`

```
public class Person {
    private String name;
    private int age;
    // ... constructor ...

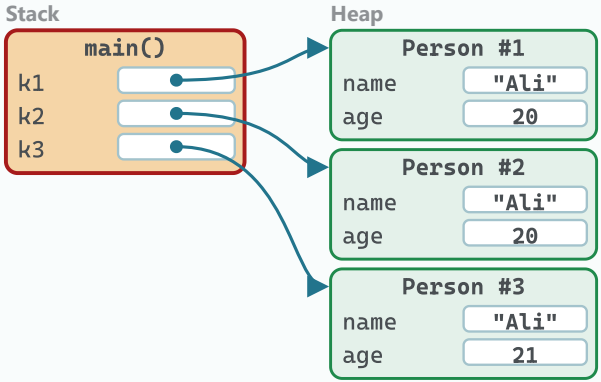
    @Override
    public boolean equals(Object obj) {
        // same reference: equal
        if (this == obj) return true;
        // null or other class: not equal
        if (obj == null
                || getClass() != obj.getClass())
            return false;
        // compare the fields
        Person p = (Person) obj;
        return age == p.age
               && name.equals(p.name);
    }
}
```

- `this == obj` : the same object returns `true` at once (fast path).
- `getClass()` gives the object's **actual** class; both sides must have exactly the same class.
- Compare primitive fields (`int age`) with `==` and object fields (`String name`) with `equals` .

Type check	Can a subclass object count as equal?
`instanceof`	Yes (a Cat object is an Animal)
`getClass()`	No (the classes must match exactly)

`equals()`: Code Trace

```
1  public boolean equals(Object obj) {
2      if (this == obj) return true;
3      if (obj == null || getClass() != obj.getClass())
4          return false;
5      Person p = (Person) obj;
6      return age == p.age && name.equals(p.name);
7  }
8  // in main:
   Person k1 = new Person("Ali", 20);
   Person k2 = new Person("Ali", 20);
   Person k3 = new Person("Ali", 21);
12 System.out.println(k1.equals(k2));
13 System.out.println(k1.equals(k3));
14 System.out.println(k1.equals(null));
```



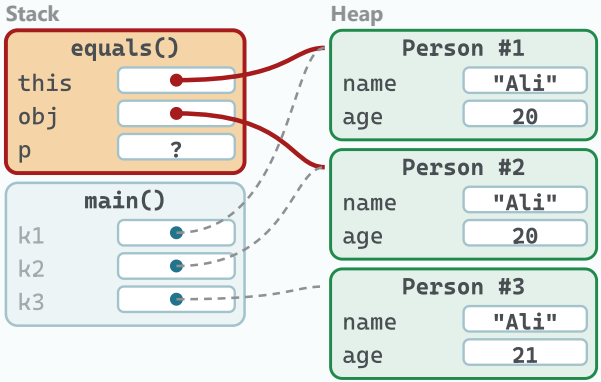
There are three separate objects. `k1` and `k2` have the same contents; `k3` differs only in age.

Step 1 / 9

OUTPUT

`equals()`: Code Trace

```
▶ public boolean equals(Object obj) {
2     if (this == obj) return true;
3     if (obj == null || getClass() != obj.getClass())
4         return false;
5     Person p = (Person) obj;
6     return age == p.age && name.equals(p.name);
7 }
8 // in main:
9 Person k1 = new Person("Ali", 20);
10 Person k2 = new Person("Ali", 20);
11 Person k3 = new Person("Ali", 21);
▶ System.out.println(k1.equals(k2));
13 System.out.println(k1.equals(k3));
14 System.out.println(k1.equals(null));
```



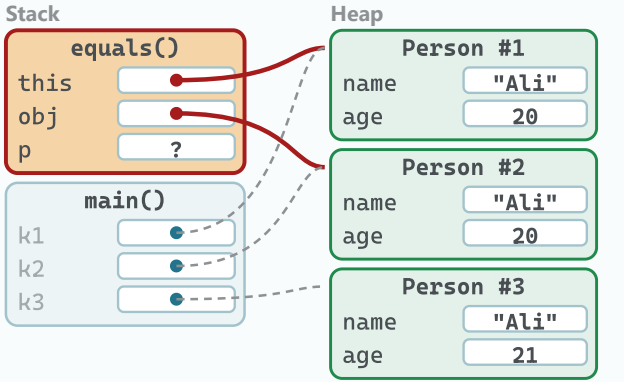
`k1.equals(k2)` : in the new frame `this` = #1 and `obj` = #2. The type of `obj` is `Object`.

Step 2 / 9

OUTPUT

`equals()`: Code Trace

```
1  public boolean equals(Object obj) {
2      if (this == obj) return true;
3      if (obj == null || getClass() != obj.getClass())
4          return false;
5      Person p = (Person) obj;
6      return age == p.age && name.equals(p.name);
7  }
8  // in main:
9  Person k1 = new Person("Ali", 20);
10 Person k2 = new Person("Ali", 20);
11 Person k3 = new Person("Ali", 21);
12 System.out.println(k1.equals(k2));
13 System.out.println(k1.equals(k3));
14 System.out.println(k1.equals(null));
```



this == obj → **false**

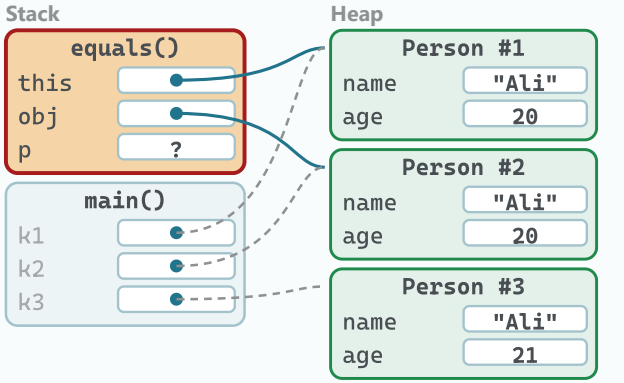
`this == obj` compares addresses: #1 ≠ #2, so the fast path does not apply.

Step 3 / 9

OUTPUT

`equals()`: Code Trace

```
1  public boolean equals(Object obj) {
2      if (this == obj) return true;
▶      if (obj == null || getClass() != obj.getClass())
4          return false;
5      Person p = (Person) obj;
6      return age == p.age && name.equals(p.name);
7  }
8  // in main:
9  Person k1 = new Person("Ali", 20);
10 Person k2 = new Person("Ali", 20);
11 Person k3 = new Person("Ali", 21);
12 System.out.println(k1.equals(k2));
13 System.out.println(k1.equals(k3));
14 System.out.println(k1.equals(null));
```



`null?` **false** · `Person ≠ Person?` **false**

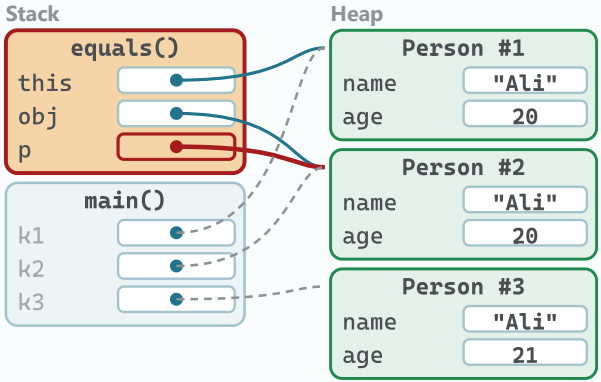
`obj` is not null and `getClass()` is `Person` on both sides. The condition is false, so `return false` is skipped.

Step 4 / 9

OUTPUT

`equals()`: Code Trace

```
1  public boolean equals(Object obj) {
2      if (this == obj) return true;
3      if (obj == null || getClass() != obj.getClass())
4          return false;
▶      Person p = (Person) obj;
6      return age == p.age && name.equals(p.name);
7  }
8  // in main:
9  Person k1 = new Person("Ali", 20);
10 Person k2 = new Person("Ali", 20);
11 Person k3 = new Person("Ali", 21);
12 System.out.println(k1.equals(k2));
13 System.out.println(k1.equals(k3));
14 System.out.println(k1.equals(null));
```



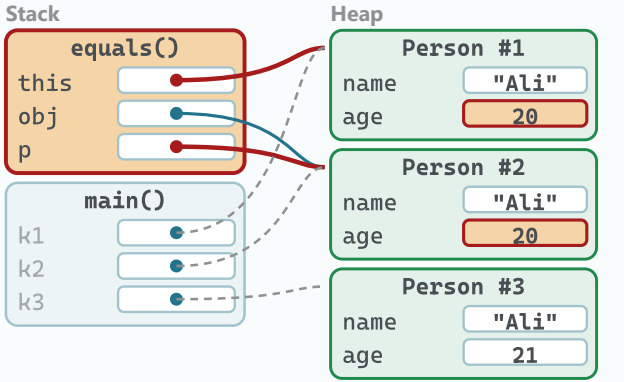
The downcast creates no new object: `p` points to the same #2, this time with type `Person`, so `p.age` can now be written.

Step 5 / 9

OUTPUT

`equals()`: Code Trace

```
1  public boolean equals(Object obj) {
2      if (this == obj) return true;
3      if (obj == null || getClass() != obj.getClass())
4          return false;
5      Person p = (Person) obj;
▶      return age == p.age && name.equals(p.name);
7  }
8  // in main:
9  Person k1 = new Person("Ali", 20);
10 Person k2 = new Person("Ali", 20);
11 Person k3 = new Person("Ali", 21);
12 System.out.println(k1.equals(k2));
13 System.out.println(k1.equals(k3));
14 System.out.println(k1.equals(null));
```



`20 == 20` **true** `&& "Ali".equals("Ali")` **true**

The primitive `age` is compared with `==`, the `String` `name` with `equals`. Both hold: **true**.

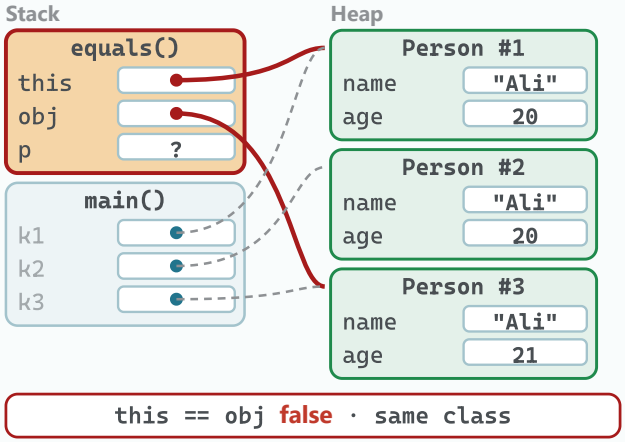
Step 6 / 9

OUTPUT

```
true
```

`equals()`: Code Trace

```
1  public boolean equals(Object obj) {
▶      if (this == obj) return true;
▶      if (obj == null || getClass() != obj.getClass())
4          return false;
5      Person p = (Person) obj;
6      return age == p.age && name.equals(p.name);
7  }
8  // in main:
9  Person k1 = new Person("Ali", 20);
10 Person k2 = new Person("Ali", 20);
11 Person k3 = new Person("Ali", 21);
12 System.out.println(k1.equals(k2));
▶  System.out.println(k1.equals(k3));
14 System.out.println(k1.equals(null));
```



`this == obj` **false** · same class

`k1.equals(k3)`: a new call with `obj` = #3. Different object, same class: on to the fields.

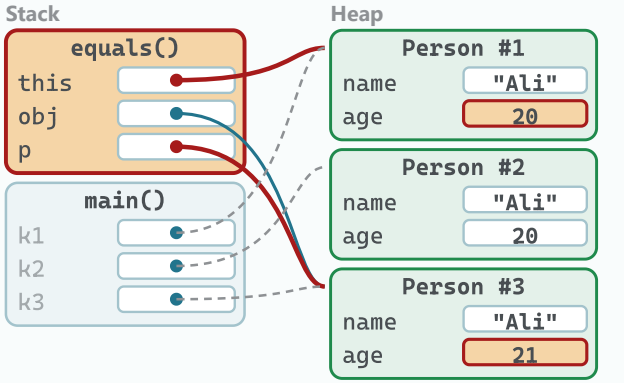
Step 7 / 9

OUTPUT

```
true
```

`equals()`: Code Trace

```
1  public boolean equals(Object obj) {
2      if (this == obj) return true;
3      if (obj == null || getClass() != obj.getClass())
4          return false;
5      Person p = (Person) obj;
▶      return age == p.age && name.equals(p.name);
7  }
8  // in main:
9  Person k1 = new Person("Ali", 20);
10 Person k2 = new Person("Ali", 20);
11 Person k3 = new Person("Ali", 21);
12 System.out.println(k1.equals(k2));
13 System.out.println(k1.equals(k3));
14 System.out.println(k1.equals(null));
```



`20 == 21` **false** → `name.equals` not called

`20 == 21` is false; `&&` short-circuits and `name.equals` is never called. The result is **false**.

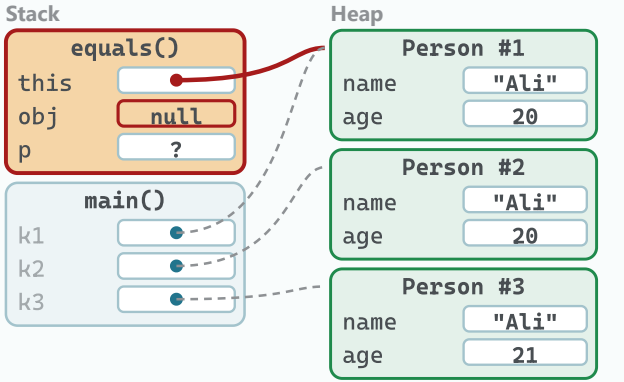
Step 8 / 9

OUTPUT

```
true
false
```

`equals()`: Code Trace

```
1  public boolean equals(Object obj) {
2      if (this == obj) return true;
▶      if (obj == null || getClass() != obj.getClass())
▶          return false;
5      Person p = (Person) obj;
6      return age == p.age && name.equals(p.name);
7  }
8  // in main:
9  Person k1 = new Person("Ali", 20);
10 Person k2 = new Person("Ali", 20);
11 Person k3 = new Person("Ali", 21);
12 System.out.println(k1.equals(k2));
13 System.out.println(k1.equals(k3));
▶  System.out.println(k1.equals(null));
```



`obj == null` true → `getClass()` skipped

`obj == null` is true, so `||` skips its right side. Had `obj.getClass()` run, it would throw a `NullPointerException`.

Step 9 / 9

OUTPUT

```
true
false
false
```

`equals` and `hashCode`

Rule: if you override `equals`, override `hashCode` too. Equal objects must have the **same** hashCode.

```
import java.util.Objects;
public class Point {
    private int x, y;
    public Point(int x, int y) {
        this.x = x; this.y = y;
    }
    @Override public boolean equals(Object o) {
        return (o instanceof Point n)
               && x == n.x && y == n.y;
    }
    @Override public int hashCode() {
        return Objects.hash(x, y);
    }
    @Override public String toString() {
        return "Point[x=" + x + ", y=" + y + "]";
    }
}
```

Why? `HashSet` and `HashMap` first pick a "bucket" by hashCode, then call `equals`. If the hashCodes differ, equal objects are never compared.

```
HashSet<Point> set = new HashSet<>();
set.add(new Point(3, 4));
set.add(new Point(3, 4));
System.out.println(set.size() + " "
    + set.contains(new Point(3, 4)));
```

In class Point	Output
only `equals`	`2 false` ✗
`equals` + `hashCode`	`1 true` ✓

The pattern `o instanceof Point n` requires Java 16+; the classic form is `if (o instanceof Point) { Point n = (Point) o; … }`.

equals and hashCode Inside a HashSet: Code Trace

```
1  public boolean equals(Object o) {
2      if (!(o instanceof Point)) return false;
3      Point n = (Point) o;
4      return x == n.x && y == n.y;
5  }
6  public int hashCode() {
7      return Objects.hash(x, y);  // 31·(31 + x) + y
8  }
9  // in main:
▶ HashSet<Point> set = new HashSet<>();
11 set.add(new Point(3, 4));
12 set.add(new Point(4, 3));
13 set.add(new Point(3, 4));
14 System.out.println(set.size());
15 System.out.println(set.contains(new Point(4, 3)));
```



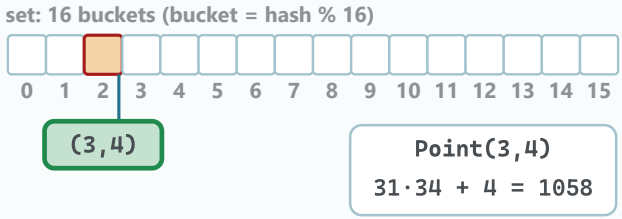
An empty `HashSet` : each element goes into one of 16 buckets, chosen by `hashCode()` .

Step 1 / 7

OUTPUT

equals and hashCode Inside a HashSet: Code Trace

```
1  public boolean equals(Object o) {
2      if (!(o instanceof Point)) return false;
3      Point n = (Point) o;
4      return x == n.x && y == n.y;
5  }
▶ public int hashCode() {
▶      return Objects.hash(x, y);  // 31·(31 + x) + y
8  }
9  // in main:
10 HashSet<Point> set = new HashSet<>();
▶ set.add(new Point(3, 4));
12 set.add(new Point(4, 3));
13 set.add(new Point(3, 4));
14 System.out.println(set.size());
15 System.out.println(set.contains(new Point(4, 3)));
```



1058 % 16 = 2 → bucket empty, added

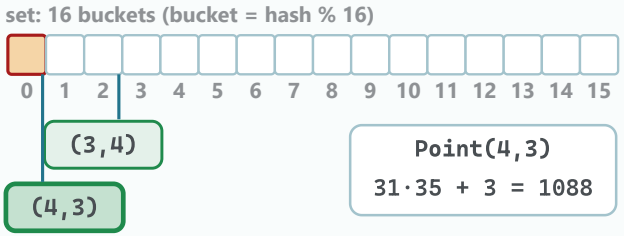
`hashCode()` = 31·(31+3)+4 = **1058**; 1058 % 16 = 2. The bucket is empty: the object is added.

Step 2 / 7

OUTPUT

equals and hashCode Inside a HashSet: Code Trace

```
1  public boolean equals(Object o) {
2      if (!(o instanceof Point)) return false;
3      Point n = (Point) o;
4      return x == n.x && y == n.y;
5  }
6  public int hashCode() {
▶      return Objects.hash(x, y);  // 31·(31 + x) + y
8  }
9  // in main:
10 HashSet<Point> set = new HashSet<>();
11 set.add(new Point(3, 4));
▶  set.add(new Point(4, 3));
13 set.add(new Point(3, 4));
14 System.out.println(set.size());
15 System.out.println(set.contains(new Point(4, 3)));
```



`1088 % 16 = 0 → bucket empty, added`

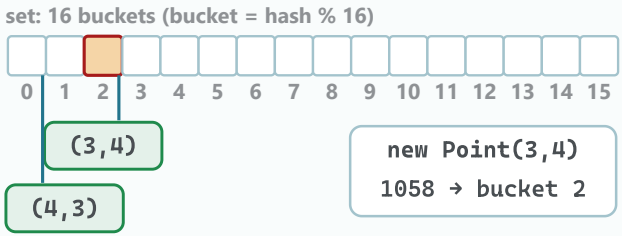
Same numbers, other order: hash **1088**, bucket 0. Again an empty bucket, so it is added.

Step 3 / 7

OUTPUT

equals and hashCode Inside a HashSet: Code Trace

```
 1 public boolean equals(Object o) {
 2     if (!(o instanceof Point)) return false;
 3     Point n = (Point) o;
 4     return x == n.x && y == n.y;
 5 }
 6 public int hashCode() {
 ▶     return Objects.hash(x, y);  // 31·(31 + x) + y
 8 }
 9 // in main:
10 HashSet<Point> set = new HashSet<>();
11 set.add(new Point(3, 4));
12 set.add(new Point(4, 3));
 ▶ set.add(new Point(3, 4));
14 System.out.println(set.size());
15 System.out.println(set.contains(new Point(4, 3)));
```



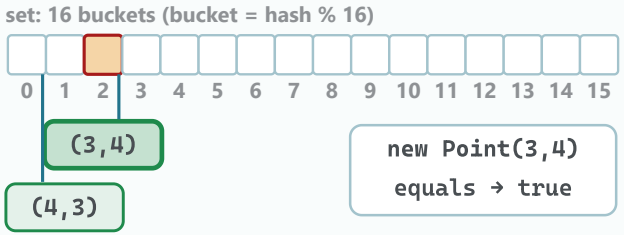
A **new** `Point(3, 4)` object: hash 1058 again, the same bucket 2. The bucket is occupied, so `equals` is asked against its contents.

Step 4 / 7

OUTPUT

equals and hashCode Inside a HashSet: Code Trace

```
▶ public boolean equals(Object o) {
2     if (!(o instanceof Point)) return false;
3     Point n = (Point) o;
▶     return x == n.x && y == n.y;
5 }
6 public int hashCode() {
7     return Objects.hash(x, y);  // 31·(31 + x) + y
8 }
9 // in main:
10 HashSet<Point> set = new HashSet<>();
11 set.add(new Point(3, 4));
12 set.add(new Point(4, 3));
13 set.add(new Point(3, 4));
14 System.out.println(set.size());
15 System.out.println(set.contains(new Point(4, 3)));
```



`3 == 3 && 4 == 4 → true: not added`

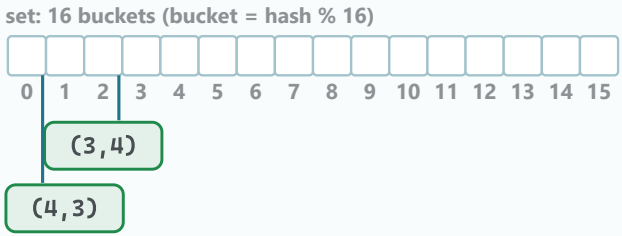
`equals` is true: the set does not take the same element twice, and `add` returns `false`.

Step 5 / 7

OUTPUT

equals and hashCode Inside a HashSet: Code Trace

```
 1 public boolean equals(Object o) {
 2     if (!(o instanceof Point)) return false;
 3     Point n = (Point) o;
 4     return x == n.x && y == n.y;
 5 }
 6 public int hashCode() {
 7     return Objects.hash(x, y);  // 31·(31 + x) + y
 8 }
 9 // in main:
10 HashSet<Point> set = new HashSet<>();
11 set.add(new Point(3, 4));
12 set.add(new Point(4, 3));
13 set.add(new Point(3, 4));
▶ System.out.println(set.size());
15 System.out.println(set.contains(new Point(4, 3)));
```



The set holds two elements: **2**.

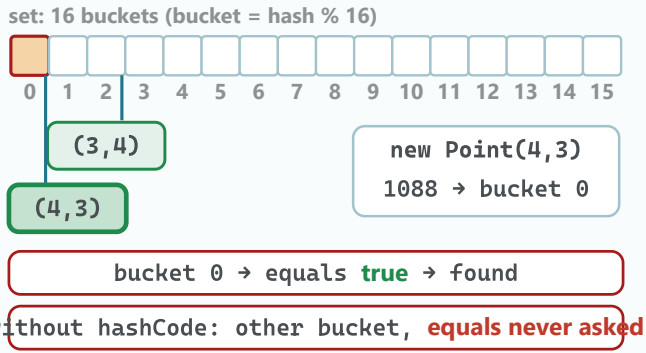
Step 6 / 7

OUTPUT

```
2
```

equals and hashCode Inside a HashSet: Code Trace

```
1  public boolean equals(Object o) {
2      if (!(o instanceof Point)) return false;
3      Point n = (Point) o;
▶      return x == n.x && y == n.y;
5  }
6  public int hashCode() {
▶      return Objects.hash(x, y);  // 31·(31 + x) + y
8  }
9  // in main:
10 HashSet<Point> set = new HashSet<>();
11 set.add(new Point(3, 4));
12 set.add(new Point(4, 3));
13 set.add(new Point(3, 4));
14 System.out.println(set.size());
▶  System.out.println(set.contains(new Point(4, 3)));
```



`contains` also finds the bucket first, then asks `equals`: **true**. Without `hashCode`, equal objects would land in different buckets.

Step 7 / 7

OUTPUT

```
2
true
```

Summary

Concept	Essence
Inheritance	`class Child extends Parent`; the subclass inherits the superclass members and adds new ones (is-a)
Not inherited	Constructors; `private` members exist in the object but cannot be reached directly
`super`	`super(...)`: superclass constructor (first line); `super.m()`: the superclass version
Constructor chain	Completes in the order Object → … → Parent → Child
Overriding	Same signature, new body in the subclass; `@Override` lets the compiler check
`protected`	Class + subclasses + same package
`final`	Method: cannot be overridden; class: cannot be extended
Has-a	Composition: a field of another class type; composition and interfaces replace multiple inheritance
Object	Ancestor of every class; override `toString`, `equals`, `hashCode`
`==` / `equals`	Compare references / compare contents

SUMMARY

Common Mistakes

Not calling super(...)

If the superclass has no no-arg constructor: `constructor Animal in class Animal cannot be applied to given types`.

Accessing a private field directly

`health += 5;` in the subclass: `health has private access in Animal`. Fix: a getter or a protected method.

Writing equals(Point)

If the parameter is not `Object`, it is an overload; with `@Override`: `method does not override or implement a method from a supertype`.

Comparing objects with ==

For two objects with the same contents `==` → `false`. Compare contents with `equals`, `String` included.

equals without hashCode

No compile error, but `HashSet` / `HashMap` misbehave: the same point is added twice.

Extending two classes

`class D extends B, C` → `'{' expected`. Fix: composition or an interface.

Narrowing access

If `public void draw()` becomes `void draw()` in the subclass: `attempting to assign weaker access privileges`.

Unchecked downcast

`(Tire) p` when p is not a Tire: `ClassCastException`. Check with `instanceof` first.

SUMMARY

Common Mistakes (2)

this(...) and super(...) together

Both must be the first statement: `call to super must be first statement in constructor`. The constructor at the end of the chain calls `super(...)`.

Changing the return type

`String reproduce()` : `return type String is not compatible with Animal`. Only a subtype (covariant) is allowed.

Thinking a private method is overridden

No error, but the superclass code still calls its own `private` version. `@Override` catches this mistake.

Reaching a protected member via a supertype reference

`h.name` in a subclass in another package: `name has protected access in Animal`. Access must go through the subclass's own type.

Widening the exception list

`void f() throws Exception` when the superclass method has no `throws` : `overridden method does not throw Exception`.

Getting field order wrong

Subclass field initializers run **after** the superclass constructor finishes; while it runs, they still hold defaults.

Inheritance just to share code

If is-a does not hold, the subclass also gets unwanted `public` methods: `Stack extends ArrayList` allows inserting in the middle. Fix: composition.

Overriding a self-calling method

Other superclass methods may call the subclass version too: the counter goes up twice. Fix: read the superclass documentation, or use composition.

SUMMARY

Practice Problems

```
class A {
    A() { System.out.println("A"); }
    void print() { System.out.println("A.print"); }
}
class B extends A {
    B() { System.out.println("B"); }
    @Override void print() { System.out.println("B.print"); }
}
class C extends B {
    C() { super(); System.out.println("C"); }
    @Override void print() {
        super.print();
        System.out.println("C.print");
    }
}
```

1. What does `C c = new C();` print?
2. What does `c.print();` print next?
3. What does `A a = new B();` `a.print();` print?
4. What is `c instanceof A` ?
5. In a `Student` class that does not override `equals`, what is `o1.equals(o2)` for two objects with the same name?

▶ **Show answers**

SUMMARY

Practice Problems (2)

```
class A {
    static { System.out.print("As "); }
    { System.out.print("Ab "); }
    A() { System.out.print("A "); }
}
class B extends A {
    static { System.out.print("Bs "); }
    { System.out.print("Bb "); }
    B() { this(0); System.out.print("B "); }
    B(int x) { System.out.print("B" + x + " "); }
}
```

1. What does the program's first `new B();` print? (Lines in plain braces are instance initializer blocks.)
2. What does `new B(7);` print next?
3. With the `Square` from the substitution example, what is `d.area()` after `Rectangle d = new Square(); d.setHeight(4); d.setWidth(5);`?
4. `Parent` has `protected Object copy()`. Does `@Override public Child copy()` compile inside `Child extends Parent`?

▶ **Show answers**

SUMMARY

Exercises

1. Write `Employee` (`name`, `salary`) and a subclass `Manager` (`bonus`); add `totalIncome()` and a `toString()` that uses `super.toString()`.
2. `Animal`'s only constructor is `Animal(String name)`. Write the constructor `Bird(String name, boolean canFly)`.
3. Write `equals` and `hashCode` for `Student` (`int no`, `String name`).
4. Can a class in another package that is **not** a subclass access a `protected` field?
5. is-a or has-a? Student–Person, Car–Wheel, Library–Book, Square–Shape.

▶ **Answers / hints**

References

- Oracle, *The Java Tutorials*: Inheritance; Overriding and Hiding Methods; Using the Keyword super; Object as a Superclass.
- J. Gosling et al., *The Java Language Specification*, Java SE 21 Edition (8.1.4 Superclasses and Subclasses, 8.4.8 Inheritance, Overriding, and Hiding).
- P. Deitel and H. Deitel, *Java: How to Program*, 11th ed., Pearson (Chapter 9: Object-Oriented Programming: Inheritance).
- K. Sierra and B. Bates, *Head First Java*, 3rd ed., O'Reilly (Chapter 7: Inheritance and Polymorphism).
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12th ed., Pearson (Chapter 5: Inheritance).
- J. Bloch, *Effective Java*, 3rd ed., Addison-Wesley (Items 10–11: equals and hashCode; Item 18: favor composition over inheritance).

The End

Chapter 4.3: Inheritance

Previous: Chapter 4.2: Encapsulation · Next: Chapter 4.4: Polymorphism