Chapter 4.4: Polymorphism

Object Oriented Programming with Java

What polymorphism is · The reference rule · Upcasting and downcasting · instanceof · Dynamic binding · Method tables · Overloading · Shape and anti-tank examples · equals and symmetry

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_04_Polymorphism.html

INTRODUCTION

Learning Outcomes

Concept

Define polymorphism and distinguish compile-time from run-time polymorphism.

The reference rule

Explain why a superclass reference can hold a subclass object, and why the reverse is a compile error.

Type conversion

Write upcasts and downcasts; prevent `ClassCastException` with `instanceof`.

Dynamic binding

Trace step by step how the method called is chosen by the object's **actual class**; show the method lookup in the hierarchy.

Overloading vs. overriding

Compare overloading with overriding; know which one the compiler picks and which one the JVM picks.

Design

Write general code against a superclass type, using a shape array and an anti-tank game; recognize the symmetry trap in `equals`.

THE IDEA OF POLYMORPHISM

Polymorphism

- Greek **poly** (many) + **morph** (form): the same expression leads to **different behavior** on different objects.
- A reference can refer to an object of its own type **or of any subclass**.
- `animal.makeSound()` prints `Woof woof!` for a dog and `Meow!` for a cat.
- We write the code **once, against the superclass**; each object answers the call **with its own version**.

The four pillars of object-oriented programming: **encapsulation**, **inheritance**, **polymorphism** and **abstraction**. Polymorphism is built on inheritance.

Remote control

The same "on" button turns on the TV, the air conditioner and the projector; each device does it in its own way.

Driver's license

One license lets you drive any make of car: the driver uses the "car" interface; the engine details belong to the maker.

Zoo

The keeper says "everyone, make a sound"; each animal makes its own.

Two Kinds of Polymorphism

	Compile time (static)	Run time (dynamic)
Mechanism	Overloading	Overriding + dynamic binding
Who chooses	The compiler (`javac`)	The JVM, while the program runs
What is checked?	Number and **declared** types of the arguments	The object's **actual** class
Needs inheritance?	No, works within one class	Yes, needs a class hierarchy

SAME NAME, DIFFERENT PARAMETERS

```
int sum(int a, int b) { ... }
double sum(double a, double b) { ... }

sum(5, 3);        // int version
sum(3.5, 2.7);    // double version
```

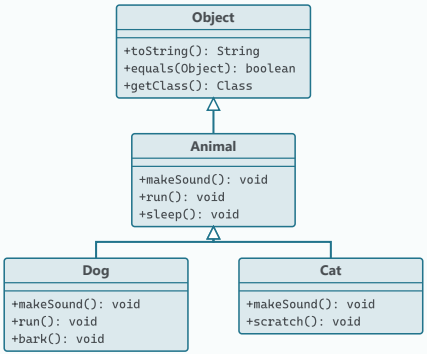
SAME SIGNATURE, DIFFERENT CLASSES

```
Animal h1 = new Dog();
Animal h2 = new Cat();

h1.makeSound(); // Woof woof!
h2.makeSound(); // Meow!
```

"Polymorphism" usually means run-time polymorphism.

Example Class Hierarchy



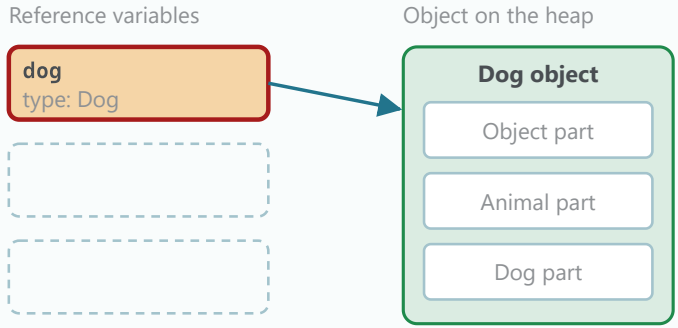

```
class Animal {
  public void makeSound() { System.out.println("Animal sound"); }
  public void run()       { System.out.println("Animal runs"); }
  public void sleep()     { System.out.println("Animal sleeps"); }
}
class Dog extends Animal {
  @Override public void makeSound() { System.out.println("Woof woof!"); }
  @Override public void run() { System.out.println("Dog runs"); }
  public void bark() { System.out.println("Dog barks"); }
}
class Cat extends Animal {
  @Override public void makeSound() { System.out.println("Meow!"); }
  public void scratch() { System.out.println("Cat scratches"); }
}
```

- `Dog` **overrides** `makeSound()` and `run()` and **adds** `bark()`; `Cat` overrides only `makeSound()` and adds `scratch()`.
- Dog and Cat **are both** Animals (is-a); either can be stored in a variable of type `Animal`.

A Reference Can Point to a Subclass Object

```
1 // reference to a Dog object
2 Dog dog = new Dog();
3 // an Animal reference can point to a Dog
4 Animal animal = dog;
5 // so can an Object reference
6 Object obj = animal;
7 System.out.println(dog == animal);
8 System.out.println(animal == obj);
9 System.out.println(obj.getClass());
```

OUTPUT



`new Dog()` creates **one** object on the heap; the `dog` reference points to it.

Step 1 / 6

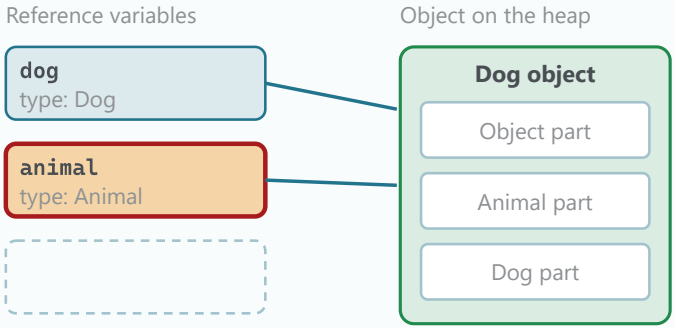
VARIABLES

`dog = → Dog object`

A Reference Can Point to a Subclass Object

```
1 // reference to a Dog object
2 Dog dog = new Dog();
3 // an Animal reference can point to a Dog
▶ Animal animal = dog;
5 // so can an Object reference
6 Object obj = animal;
7 System.out.println(dog == animal);
8 System.out.println(animal == obj);
9 System.out.println(obj.getClass());
```

OUTPUT



No new object is **created**: `animal` is a second reference to the same object. A Dog is an Animal, so the assignment is legal.

Step 2 / 6

VARIABLES

`dog = → Dog object`

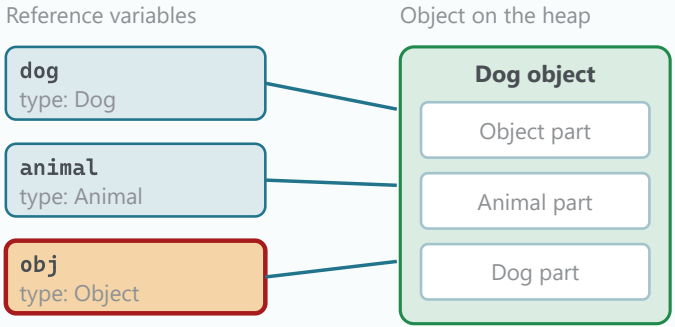
`animal = → same object`

A Reference Can Point to a Subclass Object

```
1 // reference to a Dog object
2 Dog dog = new Dog();
3 // an Animal reference can point to a Dog
4 Animal animal = dog;
5 // so can an Object reference
6 Object obj = animal;
7 System.out.println(dog == animal);
8 System.out.println(animal == obj);
9 System.out.println(obj.getClass());
```

OUTPUT





Every class extends `Object`, so an `Object` reference can point to the same object too.

Step 3 / 6

VARIABLES

`dog = → Dog object`

`animal = → same object`

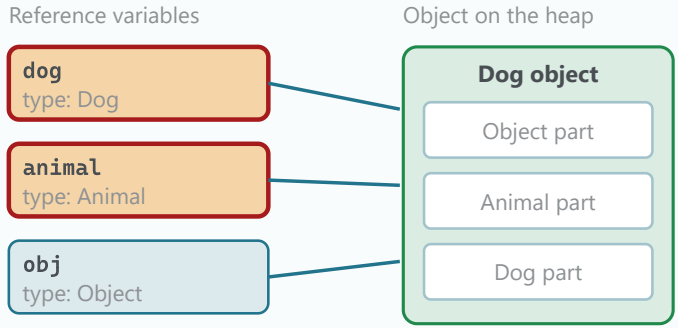
`obj = → same object`

A Reference Can Point to a Subclass Object

```
1 // reference to a Dog object
2 Dog dog = new Dog();
3 // an Animal reference can point to a Dog
4 Animal animal = dog;
5 // so can an Object reference
6 Object obj = animal;
▶ System.out.println(dog == animal);
8 System.out.println(animal == obj);
9 System.out.println(obj.getClass());
```

OUTPUT

```
true
```



`==` compares references: both point to **the same** object, so `true`.

Step 4 / 6

VARIABLES

`dog` = → **Dog object**

`animal` = → **same object**

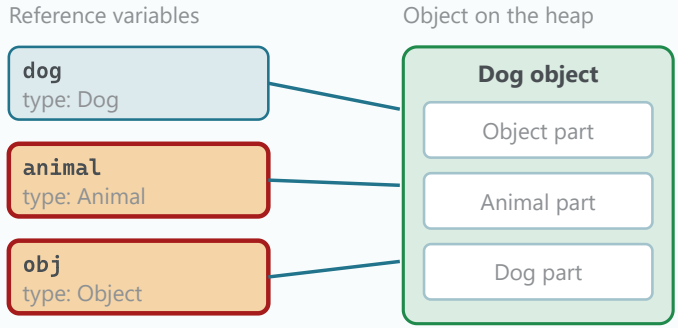
`obj` = → **same object**

A Reference Can Point to a Subclass Object

```
1 // reference to a Dog object
2 Dog dog = new Dog();
3 // an Animal reference can point to a Dog
4 Animal animal = dog;
5 // so can an Object reference
6 Object obj = animal;
7 System.out.println(dog == animal);
▶ System.out.println(animal == obj);
9 System.out.println(obj.getClass());
```

OUTPUT

```
true
true
```



Still the same object: `true`. Three references, one object.

Step 5 / 6

VARIABLES

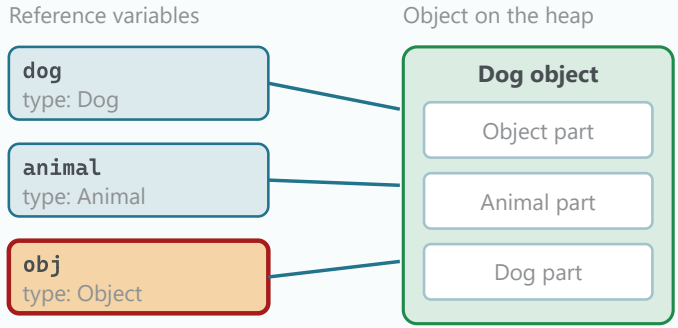
- `dog` = → **Dog object**
- `animal` = → **same object**
- `obj` = → **same object**

A Reference Can Point to a Subclass Object

```
1 // reference to a Dog object
2 Dog dog = new Dog();
3 // an Animal reference can point to a Dog
4 Animal animal = dog;
5 // so can an Object reference
6 Object obj = animal;
7 System.out.println(dog == animal);
8 System.out.println(animal == obj);
▶ System.out.println(obj.getClass());
```

OUTPUT

```
true
true
class Dog
```



The reference type is `Object`, but the object's **actual class** does not change: `Dog`.

Step 6 / 6

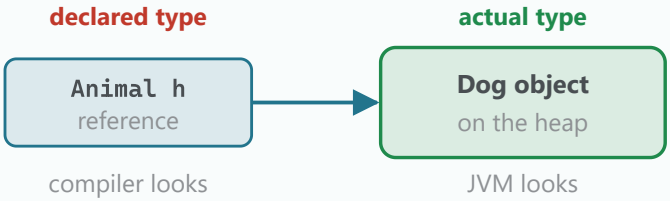
VARIABLES

`dog = → Dog object`

`animal = → same object`

`obj = → same object`

Reference Type vs. Object Type



Question	Who decides?
Can this method **be called?**	The compiler, from the **reference type**
Which version runs?	The JVM, from the **object type**
Which assignments are legal?	The compiler checks the reference type

```
Animal h = new Dog();
// declared type: Animal
// actual type:   Dog
```

```
Animal animal = new Dog();
animal.bark(); // compile error
```

```
error: cannot find symbol
  symbol:   method bark()
  location: variable animal of type Animal
```

You can't use a feature with no button on the remote: the object may be a dog, but the `Animal` "remote" has no `bark()` button.

The Reference Rule

- **An object of a subclass** can be assigned to a reference of the superclass type.
- But a superclass object **cannot** be assigned to a reference of the subclass type.

Legal

```
// Dog is a subtype of Animal
Animal animal = new Dog();
Object obj = new Cat();
```

Every dog is an animal: a dog can do everything expected of an animal.

Illegal

```
// Animal is Dog's superclass
Dog dog = new Animal();
```

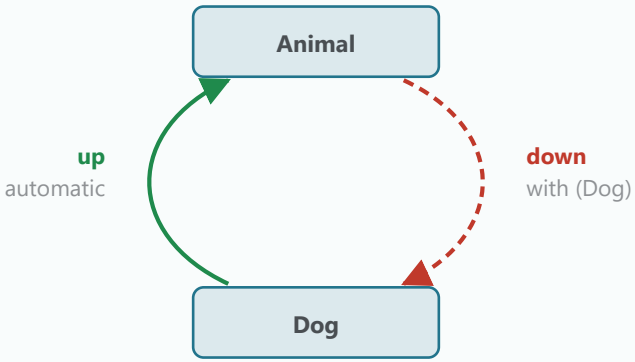
```
error: incompatible types:
  Animal cannot be converted to Dog
```

An animal may not know `bark()`; the compiler won't allow it.

`Animal h = new Dog(); Dog k = h;` gives the same error. The compiler sees the `h` reference's **declared type** (`Animal`), not the object it points to; the fix is an explicit cast.



Upcasting and Downcasting



	Upcasting	Downcasting
Direction	Sub → super	Super → sub
Syntax	Automatic (implicit)	Explicit: `(Dog) h`
Safe?	**Always**	**Depends on the object**
Also called	Widening	Narrowing

```
Dog dog = new Dog();
Animal h = dog;     // up: automatic
Animal h1 = (Animal) dog; // unnecessary
```

```
Animal h = new Dog();
Dog k = (Dog) h; // down: explicit
k.bark();        // Dog barks
```

TYPE CONVERSION

Downcasting

- A superclass reference can be **explicitly** converted to a subclass type: `(Subclass) reference`.
- Only through such a cast can a superclass reference be assigned to a subclass variable.
- Purpose: to call methods that are **not in** the superclass (that belong only to the subclass).
- A cast **does not change the object**; it only tells the compiler "treat this reference as a Dog".

```
Animal h2 = new Dog();
Dog k = (Dog) h2;
k.bark();
((Dog) h2).bark(); // in one line
```

```
Dog barks
Dog barks
```

PRECEDENCE TRAP

The dot (`.`) binds tighter than a cast. Without parentheses, `(Dog)` is applied to the **result** of `bark()` :

```
(Dog) h2.bark();    // wrong
```

```
error: not a statement
```

```
String s = (Dog) h2.name();
// h2.name() first → not in Animal
```

```
error: cannot find symbol
  symbol:   method name()
  location: variable h2 of type Animal
```

Correct: `((Dog) h2).bark();`

Bad Downcast: `ClassCastException`

```
Animal h4 = new Cat();
Dog k4 = (Dog) h4; // compiles!
k4.bark();
```

```
Exception in thread "main"
java.lang.ClassCastException:
class Cat cannot be cast to class Dog
```

```
Dog k5 = (Dog) new Animal(); // compiles!
```

```
java.lang.ClassCastException:
class Animal cannot be cast to class Dog
```

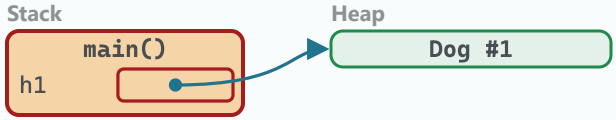
- **The compiler** allows it: a reference of type `Animal` may well point to a `Dog`.
- **The JVM** checks the object's actual class at the cast: if it is a `Cat` or a plain `Animal`, the program stops with an exception.
- So a superclass **object** can never be assigned to a subclass reference; a cast only succeeds if the object really is of that subclass.

`B objB = (B) objA;` compiles, but runs only if `objA` really points to a `B` object. If unsure, test it first with `instanceof`.

Casts between unrelated types are rejected outright: `(Cat) dog` → *incompatible types: Dog cannot be converted to Cat.*

`ClassCastException`: Code Trace

```
▶ Animal h1 = new Dog();
2 Animal h2 = new Cat();
3 Dog k1 = (Dog) h1;
4 k1.bark();
5 Dog k2 = (Dog) h2;
6 k2.bark();
7 System.out.println("Done");
```



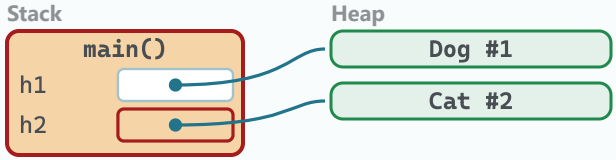
Upcast: `h1`, of type `Animal`, points to a `Dog` object.

Step 1 / 6

OUTPUT

`ClassCastException`: Code Trace

```
1 Animal h1 = new Dog();
▶ Animal h2 = new Cat();
3 Dog k1 = (Dog) h1;
4 k1.bark();
5 Dog k2 = (Dog) h2;
6 k2.bark();
7 System.out.println("Done");
```



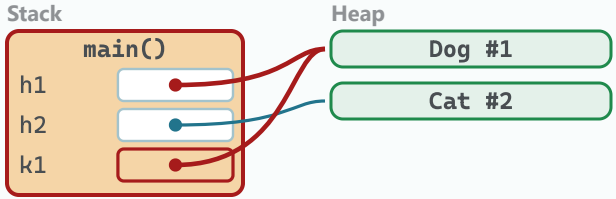
`h2` points to a `Cat`. To the compiler both are just `Animal`.

Step 2 / 6

OUTPUT

`ClassCastException`: Code Trace

```
1 Animal h1 = new Dog();
2 Animal h2 = new Cat();
▶ Dog k1 = (Dog) h1;
4 k1.bark();
5 Dog k2 = (Dog) h2;
6 k2.bark();
7 System.out.println("Done");
```



JVM check: object's class

`Dog` = **Dog** ✓

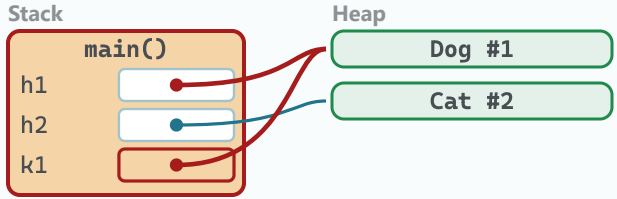
The compiler allows the cast. At run time the JVM checks the object: it really is a `Dog`. `k1` points to the same object.

Step 3 / 6

OUTPUT

`ClassCastException`: Code Trace

```
1 Animal h1 = new Dog();
2 Animal h2 = new Cat();
3 Dog k1 = (Dog) h1;
▶ k1.bark();
5 Dog k2 = (Dog) h2;
6 k2.bark();
7 System.out.println("Done");
```



Through a `Dog` reference, `bark()` can be called.

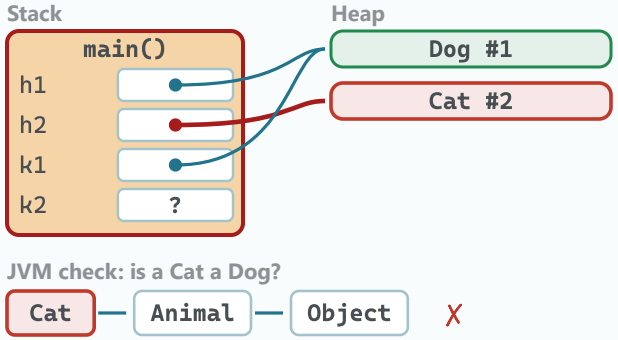
Step 4 / 6

OUTPUT

```
Dog barks
```

`ClassCastException`: Code Trace

```
1 Animal h1 = new Dog();
2 Animal h2 = new Cat();
3 Dog k1 = (Dog) h1;
4 k1.bark();
▶ Dog k2 = (Dog) h2;
6 k2.bark();
7 System.out.println("Done");
```



This line **compiles** too. But going up the chain from `Cat`, the JVM finds no `Dog`.

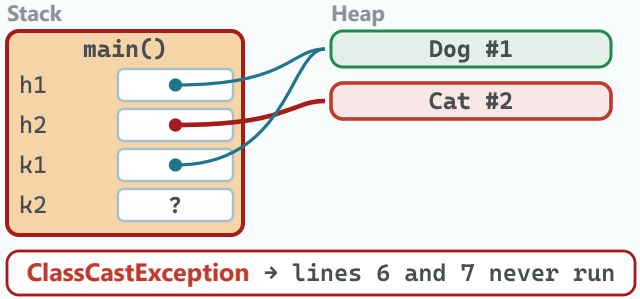
Step 5 / 6

OUTPUT

```
Dog barks
```

`ClassCastException`: Code Trace

```
1 Animal h1 = new Dog();
2 Animal h2 = new Cat();
3 Dog k1 = (Dog) h1;
4 k1.bark();
▶ Dog k2 = (Dog) h2;
6 k2.bark();
7 System.out.println("Done");
```



`ClassCastException` is thrown and nothing is assigned to `k2`; with no handler the program stops, and **Done** is never printed.

Step 6 / 6

OUTPUT

```
Dog barks
Exception in thread "main"
java.lang.ClassCastException
```

TYPE CONVERSION

`instanceof` for Safe Casting

`animals` = { `new Dog()`, `new Cat()`, `new Animal()` }

```
for (Animal x : animals) {
    if (x instanceof Dog) {
        Dog kk = (Dog) x;
        kk.bark();
    } else if (x instanceof Cat) {
        ((Cat) x).scratch();
    } else {
        System.out.println("No special behavior");
    }
}
```

```
Dog barks
Cat scratches
No special behavior
```

- `reference instanceof Type` : `true` if the object is a `Type` or a subclass of it.
- Test first, then cast: no `ClassCastException` can occur. `null instanceof Type` is always `false`.

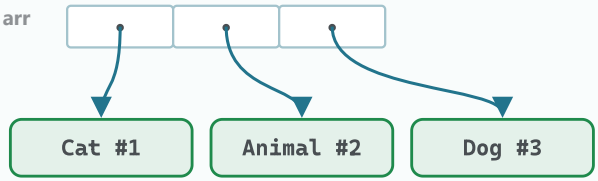
JAVA 16+: PATTERN MATCHING

```
if (x instanceof Dog kk) {
    kk.bark(); // cast is automatic
}
```

Long `instanceof` chains are usually a design smell: it is better to move the behavior into a method **overridden in the subclasses** (Exercise 5).

`instanceof` for Safe Casting: Code Trace

```
1  Animal[] arr = {new Cat(), new Animal(), new Dog()};
2  for (Animal x : arr) {
3      if (x instanceof Dog) {
4          ((Dog) x).bark();
5      } else if (x instanceof Cat) {
6          ((Cat) x).scratch();
7      } else {
8          System.out.println("No special behavior");
9      }
10 }
```



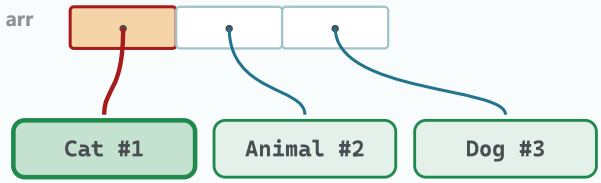
All three elements are references of type `Animal`; the objects they point to are of different classes.

Step 1 / 6

OUTPUT

`instanceof` for Safe Casting: Code Trace

```
1  Animal[] arr = {new Cat(), new Animal(), new Dog()};
▶  for (Animal x : arr) {
▶      if (x instanceof Dog) {
4          ((Dog) x).bark();
5      } else if (x instanceof Cat) {
6          ((Cat) x).scratch();
7      } else {
8          System.out.println("No special behavior");
9      }
10 }
```



x → Cat #1

`x instanceof Dog` **false**

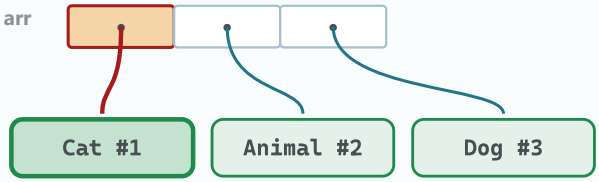
First iteration: `x` is a `Cat`. The `Dog` test is false.

Step 2 / 6

OUTPUT

`instanceof` for Safe Casting: Code Trace

```
1  Animal[] arr = {new Cat(), new Animal(), new Dog()};
2  for (Animal x : arr) {
3      if (x instanceof Dog) {
4          ((Dog) x).bark();
▶      } else if (x instanceof Cat) {
▶          ((Cat) x).scratch();
7      } else {
8          System.out.println("No special behavior");
9      }
10 }
```



x → **Cat #1**

`x instanceof Dog`	false
`x instanceof Cat`	true

The `Cat` test is true; since it was tested first, the cast `(Cat) x` is safe: **Cat scratches**.

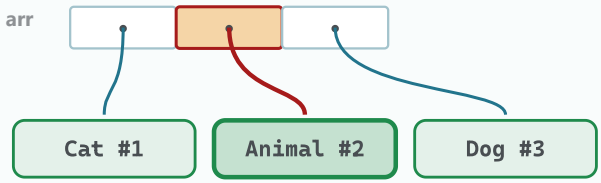
Step 3 / 6

OUTPUT

```
Cat scratches
```

`instanceof` for Safe Casting: Code Trace

```
1  Animal[] arr = {new Cat(), new Animal(), new Dog()};
▶  for (Animal x : arr) {
▶      if (x instanceof Dog) {
4          ((Dog) x).bark();
▶      } else if (x instanceof Cat) {
6          ((Cat) x).scratch();
7      } else {
8          System.out.println("No special behavior");
9      }
10 }
```



x → **Animal #2**

`x instanceof Dog`	**false**
`x instanceof Cat`	**false**

Second iteration: a plain `Animal`. Not an instance of any subclass: both tests are false.

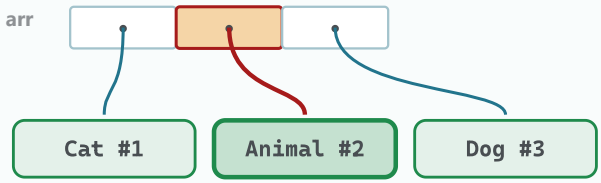
Step 4 / 6

OUTPUT

```
Cat scratches
```

`instanceof` for Safe Casting: Code Trace

```
1  Animal[] arr = {new Cat(), new Animal(), new Dog()};
2  for (Animal x : arr) {
3      if (x instanceof Dog) {
4          ((Dog) x).bark();
5      } else if (x instanceof Cat) {
6          ((Cat) x).scratch();
7      } else {
▶          System.out.println("No special behavior");
9      }
10 }
```



x → **Animal #2**

`x instanceof Dog`	false
`x instanceof Cat`	false

The `else` branch runs; no cast is attempted, so no risk.

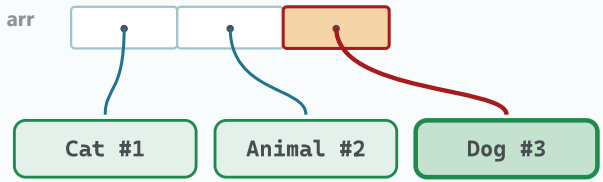
Step 5 / 6

OUTPUT

```
Cat scratches
No special behavior
```

`instanceof` for Safe Casting: Code Trace

```
1  Animal[] arr = {new Cat(), new Animal(), new Dog()};
▶  for (Animal x : arr) {
▶      if (x instanceof Dog) {
▶          ((Dog) x).bark();
5      } else if (x instanceof Cat) {
6          ((Cat) x).scratch();
7      } else {
8          System.out.println("No special behavior");
9      }
10 }
```



x → Dog #3

`x instanceof Dog` **true**

Third iteration: the `Dog` test is true and `bark()` runs. The loop ends; no `ClassCastException` was thrown in any iteration.

Step 6 / 6

OUTPUT

```
Cat scratches
No special behavior
Dog barks
```

Arrays and `ArrayStoreException`

```
Dog[] dogs = new Dog[2];
Animal[] animals = dogs;   // compiles
animals[0] = new Dog();
System.out.println("first ok");
animals[1] = new Cat();    // compiles, but...
System.out.println("never reached");
```

```
first ok
Exception in thread "main"
java.lang.ArrayStoreException: Cat
```

Same idea with a generic list

```
List<Animal> l = new ArrayList<Dog>();
```

```
error: incompatible types: ArrayList<Dog>
  cannot be converted to List<Animal>
```

- Arrays are **covariant**: a `Dog` is an `Animal`, so a `Dog[]` also counts as an `Animal[]` and the upcast compiles.
- But the array object is really a `Dog[]`. Storing a `Cat` in it through the `Animal[]` reference looks fine to the compiler; the JVM checks the element type on every store and throws `ArrayStoreException`.
- Generic collections close this hole at compile time: a `List<Dog>` is not a `List<Animal>`.
- Covariance is safe for reading (every element is an `Animal`); the problem only arises when **writing**.

Dynamic Binding

- If the reference points to a subclass object, **the subclass's** method is called.
- The version called depends not on the variable's declared type but on **the object's actual class**.
- The choice is made not at compile time but **at run time**, hence dynamic (late) binding.

```
Animal h = new Dog();
h.run();        // Dog's method
h.makeSound(); // Dog's method
h.sleep();      // not in Dog → Animal's
```

```
Dog runs
Woof woof!
Animal sleeps
```

`h.run()` IS RESOLVED IN TWO STEPS

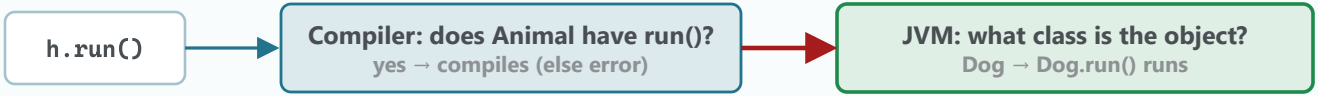
1. Compile time

The compiler looks at `h`'s type (`Animal`): does it have `run()`? **Yes** → the call compiles. Otherwise, a compile error.

2. Run time

The JVM looks at the object: its actual class is `Dog` → `Dog.run()` runs.

Rule: **the compiler asks "is there one?"**; **the JVM asks "which one?"**.



Method Lookup Starts at the Bottom of the Hierarchy

```
1 Animal h = new Dog();
2 h.run();
3 h.sleep();
4 System.out.println(h.toString());
5 // h.bark(); → compile error
```

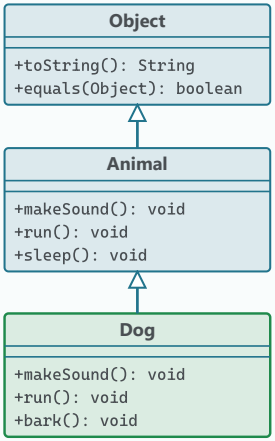
CALL STACK

`main()`

VARIABLES

`h = → Dog object`

OUTPUT



The object is a `Dog` : lookups always start **at the bottom**, at the actual class.

Method Lookup Starts at the Bottom of the Hierarchy

```
1 Animal h = new Dog();
▶ h.run();
3 h.sleep();
4 System.out.println(h.toString());
5 // h.bark(); → compile error
```

CALL STACK

`Dog.run()`

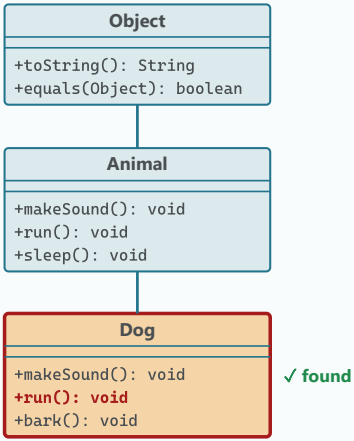
`main()`

VARIABLES

`h = → Dog object`

OUTPUT

```
Dog runs
```



`Dog` has `run()` → lookup ends, the Dog version runs.

Method Lookup Starts at the Bottom of the Hierarchy

```
1 Animal h = new Dog();
2 h.run();
▶ h.sleep();
4 System.out.println(h.toString());
5 // h.bark(); → compile error
```

CALL STACK

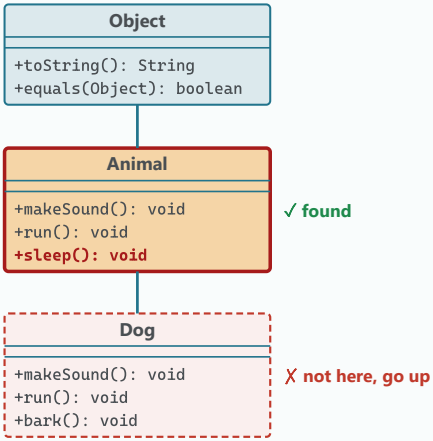
- **Animal.sleep()**
- main()

VARIABLES

h = → **Dog object**

OUTPUT

```
Dog runs
Animal sleeps
```



`Dog` has no `sleep()` → go up one level: found in `Animal`.

Method Lookup Starts at the Bottom of the Hierarchy

```
1 Animal h = new Dog();
2 h.run();
3 h.sleep();
▶ System.out.println(h.toString());
5 // h.bark(); → compile error
```

CALL STACK

`Object.toString()`

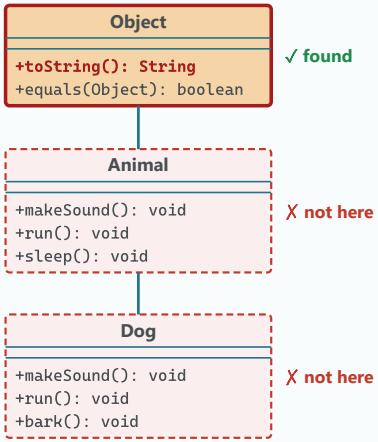
`main()`

VARIABLES

`h = → Dog object`

OUTPUT

```
Dog runs
Animal sleeps
Dog@6b884d57
```



`toString()` is found at the top, in `Object`: class name + `@` + hex hash code (varies between runs).

Method Lookup Starts at the Bottom of the Hierarchy

```
1 Animal h = new Dog();
2 h.run();
3 h.sleep();
4 System.out.println(h.toString());
▶ // h.bark(); → compile error
```

CALL STACK

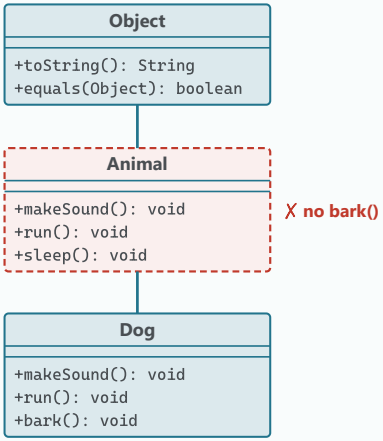
`main()`

VARIABLES

`h = → Dog object`

OUTPUT

```
Dog runs
Animal sleeps
Dog@6b884d57
```



`h.bark()` does not compile: the compiler searches up from the **reference type** `Animal` and finds nothing.

Polymorphic Array: Step-by-Step Trace

```
▶ Animal[] animals = new Animal[3];
2 animals[0] = new Dog();
3 animals[1] = new Cat();
4 animals[2] = new Animal();
5 for (int i = 0; i < animals.length; i++) {
6     animals[i].makeSound();
7 }
```

CALL STACK

`main()`

VARIABLES

`i = –`

OUTPUT





`Animal` array of 3: the elements are references, initially `null`.

Step 1 / 11

Polymorphic Array: Step-by-Step Trace

```
1 Animal[] animals = new Animal[3];
▶ animals[0] = new Dog();
3 animals[1] = new Cat();
4 animals[2] = new Animal();
5 for (int i = 0; i < animals.length; i++) {
6     animals[i].makeSound();
7 }
```

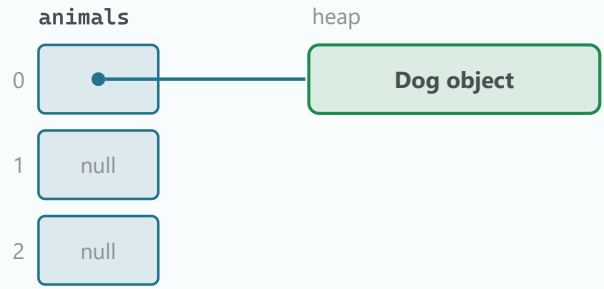
CALL STACK

`main()`

VARIABLES

`i = –`

OUTPUT



Element 0 points to a `Dog` (upcast, automatic).

Polymorphic Array: Step-by-Step Trace

```
1 Animal[] animals = new Animal[3];
2 animals[0] = new Dog();
▶ animals[1] = new Cat();
4 animals[2] = new Animal();
5 for (int i = 0; i < animals.length; i++) {
6     animals[i].makeSound();
7 }
```

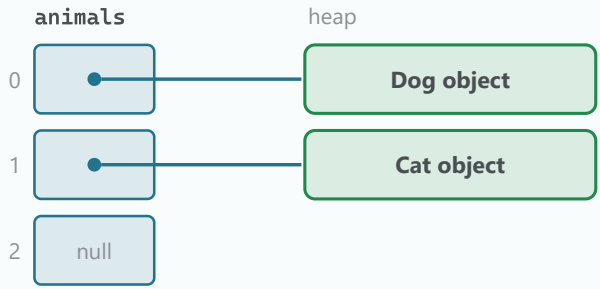
CALL STACK

`main()`

VARIABLES

`i = –`

OUTPUT



Element 1 points to a `Cat`.

Step 3 / 11

Polymorphic Array: Step-by-Step Trace

```
1 Animal[] animals = new Animal[3];
2 animals[0] = new Dog();
3 animals[1] = new Cat();
▶ animals[2] = new Animal();
5 for (int i = 0; i < animals.length; i++) {
6     animals[i].makeSound();
7 }
```

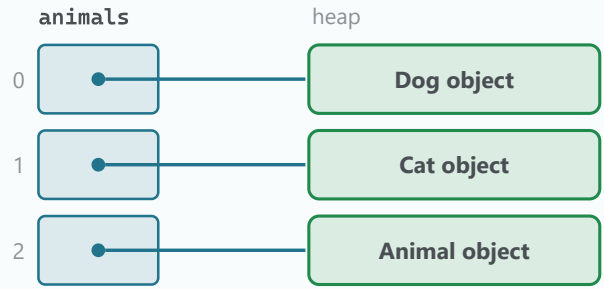
CALL STACK

`main()`

VARIABLES

`i = –`

OUTPUT



Element 2 is a plain `Animal` object. The array now holds three different forms.

Polymorphic Array: Step-by-Step Trace

```
1 Animal[] animals = new Animal[3];
2 animals[0] = new Dog();
3 animals[1] = new Cat();
4 animals[2] = new Animal();
▶ for (int i = 0; i < animals.length; i++) {
6     animals[i].makeSound();
7 }
```

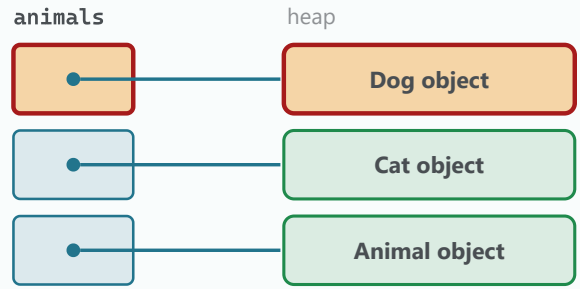
CALL STACK

`main()`

VARIABLES

`i = 0`

OUTPUT



`i = 0` , `0 < 3` true → loop body.

Polymorphic Array: Step-by-Step Trace

```
1 Animal[] animals = new Animal[3];
2 animals[0] = new Dog();
3 animals[1] = new Cat();
4 animals[2] = new Animal();
5 for (int i = 0; i < animals.length; i++) {
▶     animals[i].makeSound();
7 }
```

CALL STACK

`Dog.makeSound()`

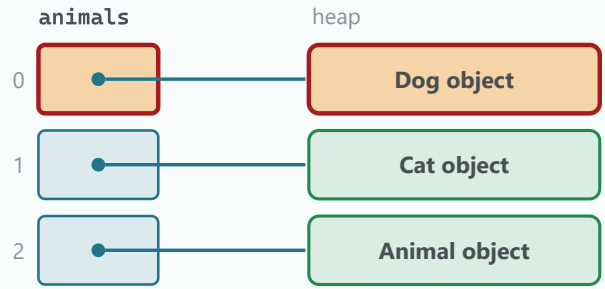
`main()`

VARIABLES

`i = 0`

OUTPUT

```
Woof woof!
```



Same line, different object: the JVM checks the object's class → `Dog.makeSound()`.

Polymorphic Array: Step-by-Step Trace

```
1 Animal[] animals = new Animal[3];
2 animals[0] = new Dog();
3 animals[1] = new Cat();
4 animals[2] = new Animal();
▶ for (int i = 0; i < animals.length; i++) {
6     animals[i].makeSound();
7 }
```

CALL STACK

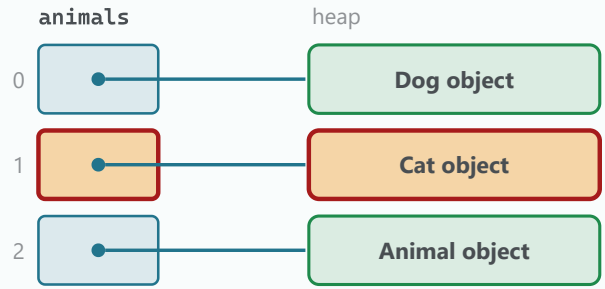
`main()`

VARIABLES

`i = 1`

OUTPUT

```
Woof woof!
```



`i = 1` , `1 < 3` true → loop body.

Polymorphic Array: Step-by-Step Trace

```
1 Animal[] animals = new Animal[3];
2 animals[0] = new Dog();
3 animals[1] = new Cat();
4 animals[2] = new Animal();
5 for (int i = 0; i < animals.length; i++) {
▶     animals[i].makeSound();
7 }
```

CALL STACK

`Cat.makeSound()`

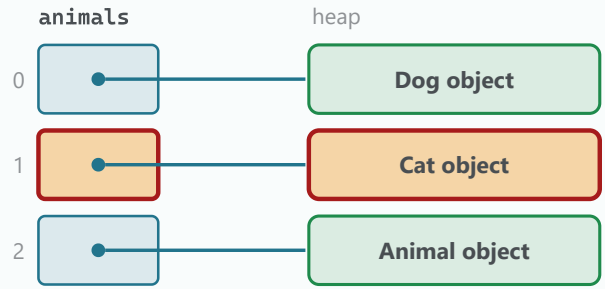
`main()`

VARIABLES

`i = 1`

OUTPUT

```
Woof woof!
Meow!
```



Same line, different object: the JVM checks the object's class → `Cat.makeSound()`.

Polymorphic Array: Step-by-Step Trace

```
1 Animal[] animals = new Animal[3];
2 animals[0] = new Dog();
3 animals[1] = new Cat();
4 animals[2] = new Animal();
▶ for (int i = 0; i < animals.length; i++) {
6     animals[i].makeSound();
7 }
```

CALL STACK

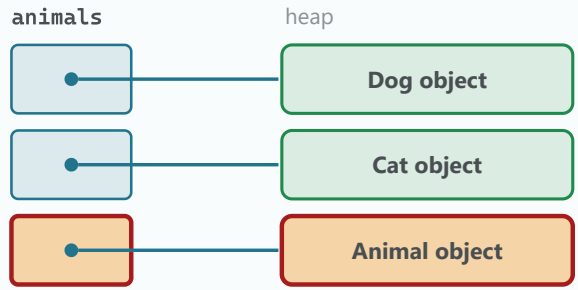
`main()`

VARIABLES

`i = 2`

OUTPUT

```
Woof woof!
Meow!
```



`i = 2` , `2 < 3` true → loop body.

Polymorphic Array: Step-by-Step Trace

```
1 Animal[] animals = new Animal[3];
2 animals[0] = new Dog();
3 animals[1] = new Cat();
4 animals[2] = new Animal();
5 for (int i = 0; i < animals.length; i++) {
▶     animals[i].makeSound();
7 }
```

CALL STACK

`Animal.makeSound()`

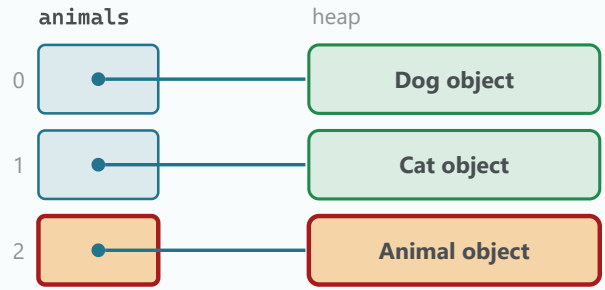
`main()`

VARIABLES

`i = 2`

OUTPUT

```
Woof woof!
Meow!
Animal sound
```



Same line, different object: the JVM checks the object's class → `Animal.makeSound()`.

Polymorphic Array: Step-by-Step Trace

```
1 Animal[] animals = new Animal[3];
2 animals[0] = new Dog();
3 animals[1] = new Cat();
4 animals[2] = new Animal();
▶ for (int i = 0; i < animals.length; i++) {
6     animals[i].makeSound();
7 }
```

CALL STACK

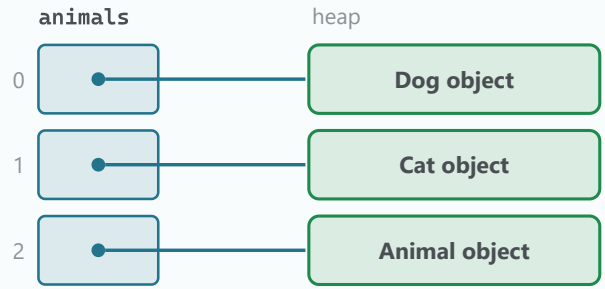
`main()`

VARIABLES

`i = 3`

OUTPUT

```
Woof woof!
Meow!
Animal sound
```



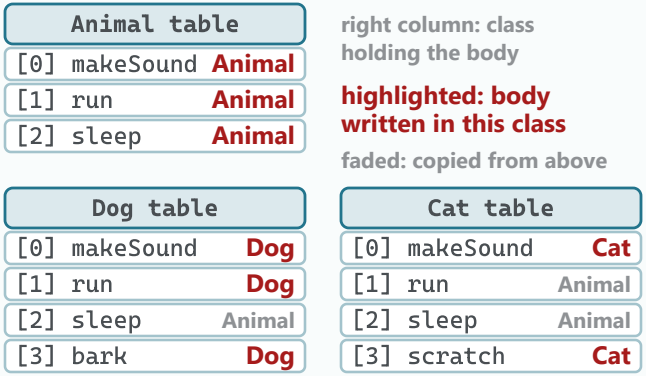
`i = 3`, `3 < 3` false → the loop ends. One line of code, three behaviors.

Behind the Scenes: The Method Table

- JVMs usually implement dynamic binding with a per-class **method table** (virtual method table): each overridable method has a fixed index (slot).
- A subclass's table starts as a copy of its superclass's; overridden slots are **replaced** with the new body, and new methods are **appended**.
- Every object knows its class. For `h.run()` the JVM goes object → class → the table's `run` slot → body. The lookup is one array access; no walking up the hierarchy.

`javac -c Main` output

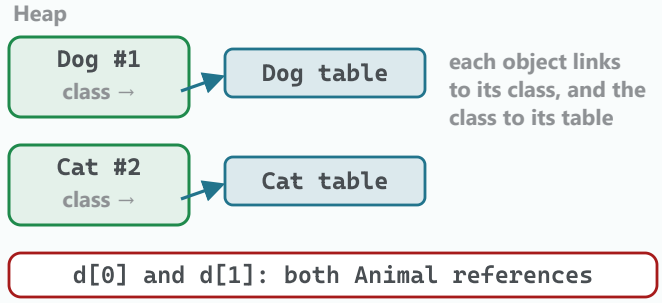
```
47: invokevirtual #15  // Method Animal.run:()V
52: invokevirtual #18  // Method Animal.makeSound:()V
```



The compiler writes only `Animal.run` into the bytecode; `invokevirtual` picks the body from the object's table. Constructors and `super.m()` use `invokespecial`, `static` methods use `invokestatic`: no table lookup. `private` methods use `invokevirtual` since Java 11, but have only one candidate.

Calls via the Method Table: Code Trace

```
1 Animal[] d = {new Dog(), new Cat()};
2 for (Animal h : d) {
3     h.run();
4     h.makeSound();
5 }
```



The two objects in the array belong to different classes; each carries a link to its own class's table.

Step 1 / 5

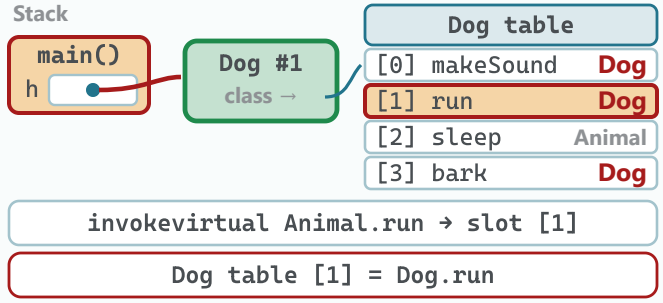
OUTPUT

VARIABLES

—

Calls via the Method Table: Code Trace

```
1 Animal[] d = {new Dog(), new Cat()};
2 for (Animal h : d) {
▶     h.run();
4     h.makeSound();
5 }
```



The bytecode says "`Animal.run`, slot 1". The object is a `Dog`: slot 1 of its table holds `Dog.run`.

Step 2 / 5

OUTPUT

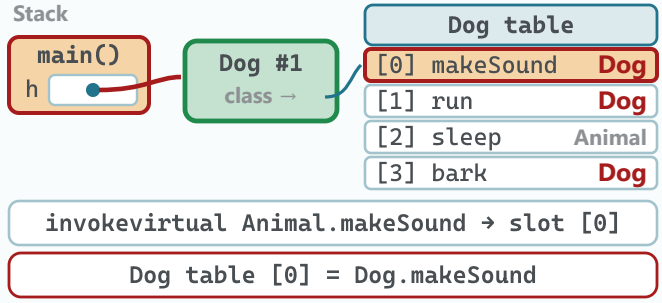
```
Dog runs
```

VARIABLES

h = →Dog #1

Calls via the Method Table: Code Trace

```
1 Animal[] d = {new Dog(), new Cat()};
2 for (Animal h : d) {
3     h.run();
▶     h.makeSound();
5 }
```



Same path, slot 0: `Woof woof!`

Step 3 / 5

OUTPUT

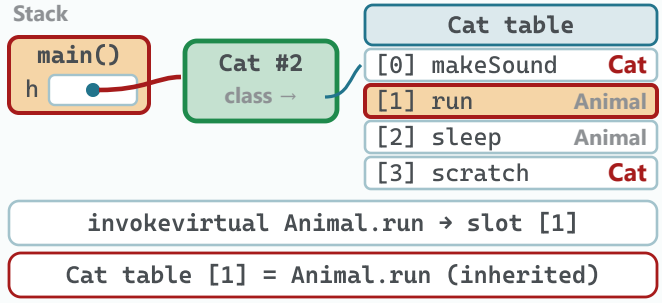
```
Dog runs
Woof woof!
```

VARIABLES

h = →Dog #1

Calls via the Method Table: Code Trace

```
1 Animal[] d = {new Dog(), new Cat()};
2 for (Animal h : d) {
▶     h.run();
4     h.makeSound();
5 }
```



Same bytecode, different object. `Cat` does not override `run`, so slot 1 of its table points to `Animal.run`.

OUTPUT

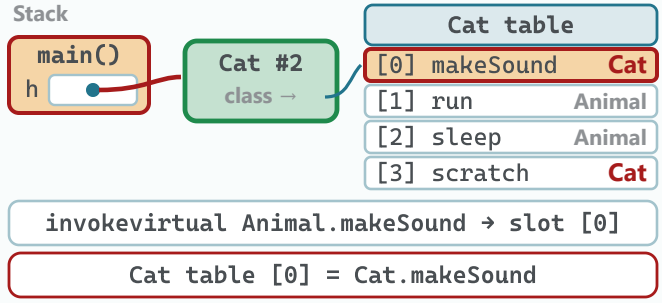
```
Dog runs
Woof woof!
Animal runs
```

VARIABLES

h = →Cat #2

Calls via the Method Table: Code Trace

```
1 Animal[] d = {new Dog(), new Cat()};
2 for (Animal h : d) {
3     h.run();
▶     h.makeSound();
5 }
```



Slot 0 again: `Meow!` None of the four calls used `if` or a type test: the tables made the choice.

OUTPUT

```
Dog runs
Woof woof!
Animal runs
Meow!
```

VARIABLES

h = →Cat #2

What Is Not Dynamically Bound?

```java
class Animal {
    String kind = "animal";

    static void info() {
        System.out.println("Animal.info()");
    }

    void introduce() {
        System.out.println("I am a " + kind);
    }
}

class Dog extends Animal {
    String kind = "dog"; // hides it

    static void info() {
        System.out.println("Dog.info()");
    }

    @Override
    void introduce() {
        System.out.println("I am a " + kind);
    }
}
```

```java
Animal h = new Dog();
System.out.println(h.kind);
h.info();
h.introduce();
```

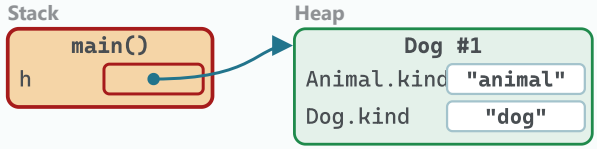
```
animal
Animal.info()
I am a dog
```

Member	Chosen by
Instance method	**Object** type (dynamic)
Field	Reference type (static)
`static` method	Reference type (static)
`private` / `final`	Cannot be overridden; static

Call a static method through the class name: `Dog.info()`. Avoid declaring a field with the same name in a subclass.

Field Hiding and `static`: Code Trace

```
1  class Animal {
2      String kind = "animal";
3      static String info() { return "Animal.info"; }
4      String introduce() { return "I am a " + kind; }
5  }
6  class Dog extends Animal {
7      String kind = "dog";              // hides
8      static String info() { return "Dog.info"; }
9      String introduce() { return "I am a " + kind; }
10 }
▶  Animal h = new Dog();   // main
12 System.out.println(h.kind);
13 System.out.println(h.info());
14 System.out.println(h.introduce());
15 System.out.println(((Dog) h).kind);
```



one object, two separate kind fields

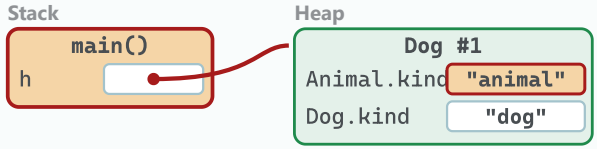
The subclass's `kind` does not override the superclass's, it **hides** it: the object has two separate fields.

Step 1 / 5

OUTPUT

Field Hiding and `static`: Code Trace

```
1  class Animal {
2      String kind = "animal";
3      static String info() { return "Animal.info"; }
4      String introduce() { return "I am a " + kind; }
5  }
6  class Dog extends Animal {
7      String kind = "dog";              // hides
8      static String info() { return "Dog.info"; }
9      String introduce() { return "I am a " + kind; }
10 }
11 Animal h = new Dog();   // main
▶  System.out.println(h.kind);
13 System.out.println(h.info());
14 System.out.println(h.introduce());
15 System.out.println(((Dog) h).kind);
```



`field → declared type (Animal)`

Fields are not dynamically bound: `h`'s declared type is `Animal`, so `Animal.kind` is read: **animal**.

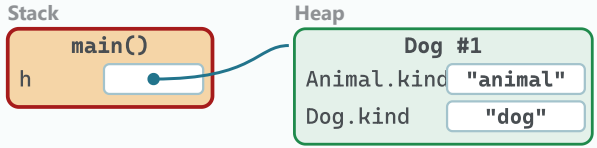
Step 2 / 5

OUTPUT

```
animal
```

Field Hiding and `static`: Code Trace

```
1  class Animal {
2      String kind = "animal";
▶      static String info() { return "Animal.info"; }
4      String introduce() { return "I am a " + kind; }
5  }
6  class Dog extends Animal {
7      String kind = "dog";              // hides
8      static String info() { return "Dog.info"; }
9      String introduce() { return "I am a " + kind; }
10 }
11 Animal h = new Dog();   // main
12 System.out.println(h.kind);
▶  System.out.println(h.info());
14 System.out.println(h.introduce());
15 System.out.println(((Dog) h).kind);
```



`static` method → declared type; object ignored

`static` methods belong to the class; the compiler binds `h.info()` to `Animal.info()`.

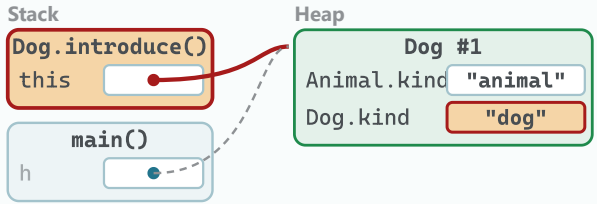
Step 3 / 5

OUTPUT

```
animal
Animal.info
```

Field Hiding and `static`: Code Trace

```
 1 class Animal {
 2     String kind = "animal";
 3     static String info() { return "Animal.info"; }
 4     String introduce() { return "I am a " + kind; }
 5 }
 6 class Dog extends Animal {
 7     String kind = "dog";              // hides
 8     static String info() { return "Dog.info"; }
 ▶     String introduce() { return "I am a " + kind; }
10 }
11 Animal h = new Dog();   // main
12 System.out.println(h.kind);
13 System.out.println(h.info());
 ▶ System.out.println(h.introduce());
15 System.out.println(((Dog) h).kind);
```



instance method → actual type (Dog)

Instance methods are dynamically bound: `Dog.introduce()` runs, and inside it `kind` is `Dog`'s field: **I am a dog**.

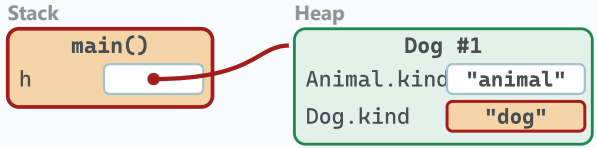
Step 4 / 5

OUTPUT

```
animal
Animal.info
I am a dog
```

Field Hiding and `static`: Code Trace

```
1  class Animal {
2      String kind = "animal";
3      static String info() { return "Animal.info"; }
4      String introduce() { return "I am a " + kind; }
5  }
6  class Dog extends Animal {
7      String kind = "dog";              // hides
8      static String info() { return "Dog.info"; }
9      String introduce() { return "I am a " + kind; }
10 }
11 Animal h = new Dog();   // main
12 System.out.println(h.kind);
13 System.out.println(h.info());
14 System.out.println(h.introduce());
▶  System.out.println(((Dog) h).kind);
```



`(Dog) h → reads Dog.kind`

The cast makes the declared type `Dog`; the object's other field is read: **dog**. Because of this confusion, subclasses should not declare a field with the same name.

Step 5 / 5

OUTPUT

```
animal
Animal.info
I am a dog
dog
```

Calling an Overridable Method from a Constructor

- Dynamic binding works even while the object is **being constructed**: the object is a `Label`, so the `draw()` call in the superclass constructor runs `Label.draw()`.
- But at that moment the subclass field initializers have not run yet: `text` is still `null`. The subclass method runs on a half-built object.
- Had the method used the field rather than just printing it, the program would have crashed.
- Rule: call only `private`, `final` or `static` methods from a constructor; they cannot be overridden.

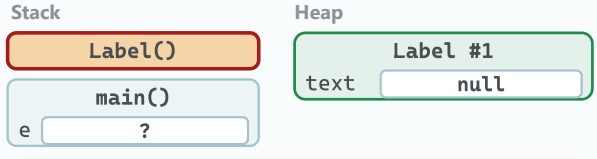
If `draw()` used `text.length()`

```
Exception in thread "main"
java.lang.NullPointerException: Cannot invoke
"String.length()" because "this.text" is null
        at Label.draw(Main.java:21)
        at Component.<init>(Main.java:16)
        at Label.<init>(Main.java:19)
```

The stack trace tells the story: `Label` constructor → `Component` constructor → back to `Label`'s method. `<init>` is the constructor's name in bytecode.

Call in a Constructor: Code Trace

```
1  class Component {
2      Component() {
3          System.out.println("Building Component");
4          draw();
5      }
6      void draw() { System.out.println("empty"); }
7  }
8  class Label extends Component {
9      private String text = "Hello";
▶      Label() { System.out.println("Label built"); }
11     @Override void draw() {
12         System.out.println("Label: " + text);
13     }
14 }
▶  Label e = new Label();   // main
16 e.draw();
```



`object allocated; Label() → implicit super()`

`new` allocates the object; `text` is `null` by default. The first thing `Label()` does is the implicit `super()` call.

Step 1 / 6

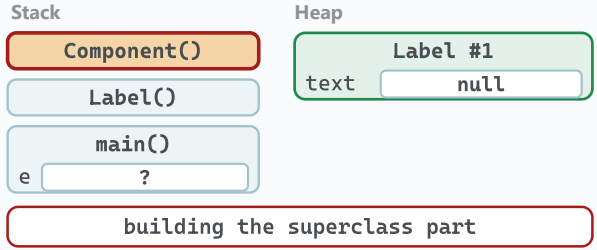
OUTPUT

VARIABLES

`text = null`

Call in a Constructor: Code Trace

```
1  class Component {
2      Component() {
▶          System.out.println("Building Component");
4          draw();
5      }
6      void draw() { System.out.println("empty"); }
7  }
8  class Label extends Component {
9      private String text = "Hello";
10     Label() { System.out.println("Label built"); }
11     @Override void draw() {
12         System.out.println("Label: " + text);
13     }
14 }
15 Label e = new Label();   // main
16 e.draw();
```



`Component()` 's body prints the first line.

Step 2 / 6

OUTPUT

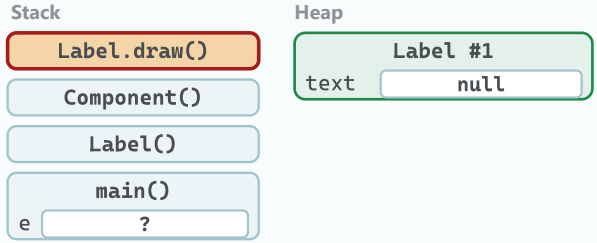
```
Building Component
```

VARIABLES

`text = null`

Call in a Constructor: Code Trace

```
1  class Component {
2      Component() {
3          System.out.println("Building Component");
▶          draw();
5      }
6      void draw() { System.out.println("empty"); }
7  }
8  class Label extends Component {
9      private String text = "Hello";
10     Label() { System.out.println("Label built"); }
11     @Override void draw() {
▶          System.out.println("Label: " + text);
13     }
14 }
15 Label e = new Label();   // main
16 e.draw();
```



`draw()` = `this.draw()`. The object's class is `Label`, so the overridden version is chosen; but `text` is not assigned yet: `Label: null`.

Step 3 / 6

OUTPUT

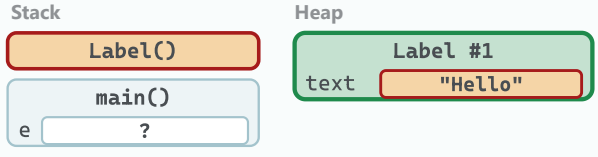
```
Building Component
Label: null
```

VARIABLES

`text = null`

Call in a Constructor: Code Trace

```
1  class Component {
2      Component() {
3          System.out.println("Building Component");
4          draw();
5      }
6      void draw() { System.out.println("empty"); }
7  }
8  class Label extends Component {
▶       private String text = "Hello";
10     Label() { System.out.println("Label built"); }
11     @Override void draw() {
12         System.out.println("Label: " + text);
13     }
14 }
15 Label e = new Label();   // main
16 e.draw();
```



super() returned → field initializers

After the superclass constructor returns, the subclass field initializer runs: `text = "Hello"`. Too late: `draw()` was already called.

Step 4 / 6

OUTPUT

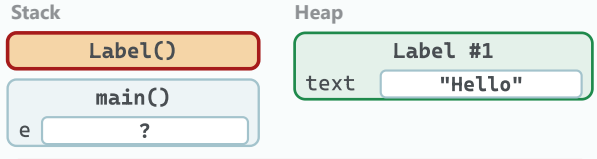
```
Building Component
Label: null
```

VARIABLES

`text = "Hello"`

Call in a Constructor: Code Trace

```
1  class Component {
2      Component() {
3          System.out.println("Building Component");
4          draw();
5      }
6      void draw() { System.out.println("empty"); }
7  }
8  class Label extends Component {
9      private String text = "Hello";
▶      Label() { System.out.println("Label built"); }
11     @Override void draw() {
12         System.out.println("Label: " + text);
13     }
14 }
15 Label e = new Label();   // main
16 e.draw();
```



The `Label()` body runs; the object is now built.

Step 5 / 6

OUTPUT

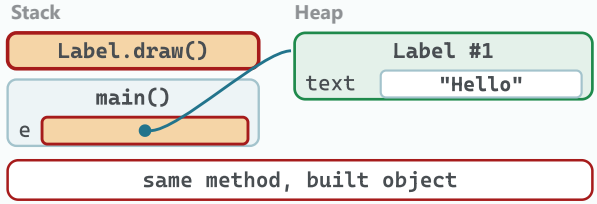
```
Building Component
Label: null
Label built
```

VARIABLES

`text = "Hello"`

Call in a Constructor: Code Trace

```
1  class Component {
2      Component() {
3          System.out.println("Building Component");
4          draw();
5      }
6      void draw() { System.out.println("empty"); }
7  }
8  class Label extends Component {
9      private String text = "Hello";
10     Label() { System.out.println("Label built"); }
11     @Override void draw() {
▶          System.out.println("Label: " + text);
13     }
14 }
15 Label e = new Label();   // main
▶ e.draw();
```



On a fully built object the same method gives the right result. The only difference is **when** the call is made.

Step 6 / 6

OUTPUT

```
Building Component
Label: null
Label built
Label: Hello
```

VARIABLES

`text = "Hello"`

Overloading Is Resolved at Compile Time

```
static void print(Animal h) {
    System.out.println("print(Animal)");
}

static void print(Dog k) {
    System.out.println("print(Dog)");
}

public static void main(String[] args) {
    Dog dog = new Dog();
    Animal animal = dog;
    print(dog);
    print(animal);
    print((Dog) animal);
}
```

```
print(Dog)
print(Animal)
print(Dog)
```

- `animal` points to a `Dog`, yet `print(animal)` → **`print(Animal)`**.
- The compiler picks among overloads by the argument's **declared type**.
- A cast changes the declared type: `(Dog) animal` → `print(Dog)`.

	Overloading	**Overriding**
Where?	Same class	Subclass
Signature	Different parameters	Same signature
Chosen at	The compiler	The JVM
Based on	Declared type	Actual type

Overloading: Most Specific Version and Ambiguity

```
static void show(Object o) { System.out.println("show(Object)"); }
static void show(Animal h) { System.out.println("show(Animal)"); }
static void show(String s) { System.out.println("show(String)"); }
// in main:
show(new Cat());
show("a");
Object o = "a";
show(o);
show((Animal) null);
```

```
show(Animal)
show(String)
show(Object)
show(Animal)
```

`show(null);`

```
error: reference to show is ambiguous
  both method show(Animal) in Main and
  method show(String) in Main match
```

- If several versions apply, the compiler picks the **most specific** one: `Cat` fits both `Animal` and `Object`; `Animal` is more specific.
- `null` fits every reference type. `Animal` and `String` are not related by inheritance, so neither is more specific: an **ambiguous** call, a compile error.
- If the applicable versions lie on one chain (`Object`, `Animal`, `Cat`) there is no ambiguity: `show(null)` picks the lowest, `show(Cat)`.
- The fix is to set the declared type with an explicit cast: `show((Animal) null)`.

Compiler or JVM? Code Trace

```
1  static void print(Animal h) {
2      System.out.print("print(Animal): ");
3      h.makeSound();
4  }
5  static void print(Dog k) {
6      System.out.print("print(Dog): ");
7      k.makeSound();
8  }
9  // in main:
▶  Animal animal = new Dog();
▶  Animal other = new Cat();
12 print(animal);
13 print(other);
14 print((Dog) animal);
```

The compiler looks at declared type	The JVM looks at object's class
`print(Animal)`	`Dog.makeSound()`
`print(Dog)`	`Cat.makeSound()`
compile time	run time

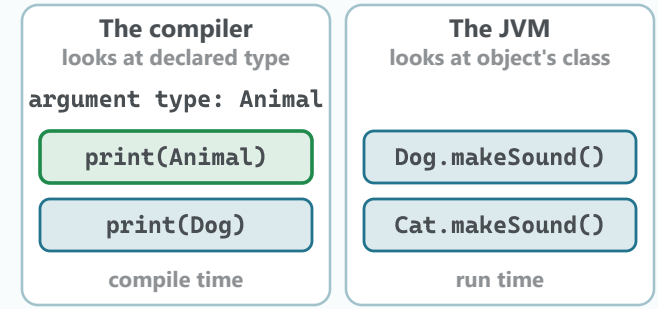
Both variables have the same declared type (`Animal`); the objects differ: `Dog` and `Cat` .

Step 1 / 7

OUTPUT

Compiler or JVM? Code Trace

```
▶ static void print(Animal h) {
▶     System.out.print("print(Animal): ");
3     h.makeSound();
4 }
5 static void print(Dog k) {
6     System.out.print("print(Dog): ");
7     k.makeSound();
8 }
9 // in main:
10 Animal animal = new Dog();
11 Animal other = new Cat();
▶ print(animal);
13 print(other);
14 print((Dog) animal);
```



`animal` points to a `Dog`, but the compiler knows only the declared type: `print(Animal)`.

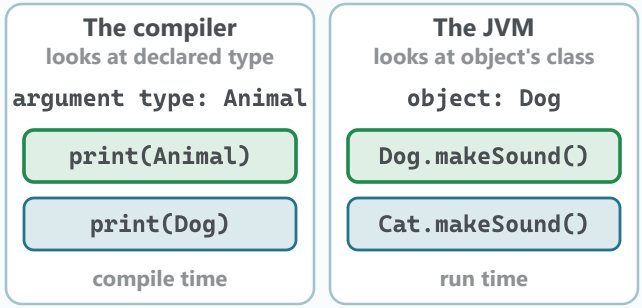
Step 2 / 7

OUTPUT

```
print(Animal):
```

Compiler or JVM? Code Trace

```
1  static void print(Animal h) {
2      System.out.print("print(Animal): ");
▶      h.makeSound();
4  }
5  static void print(Dog k) {
6      System.out.print("print(Dog): ");
7      k.makeSound();
8  }
9  // in main:
10 Animal animal = new Dog();
11 Animal other = new Cat();
12 print(animal);
13 print(other);
14 print((Dog) animal);
```



Inside the method, `h.makeSound()` is an overridden method: the JVM looks at the object: **Woof woof!**

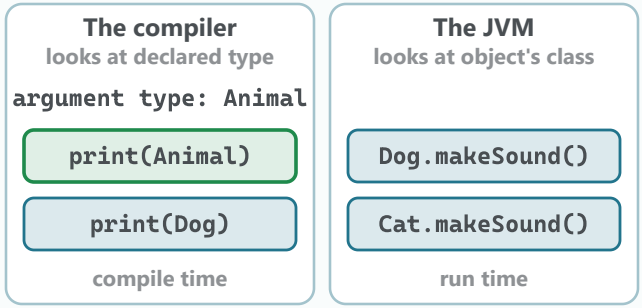
Step 3 / 7

OUTPUT

```
print(Animal): Woof woof!
```

Compiler or JVM? Code Trace

```
▶ static void print(Animal h) {
▶     System.out.print("print(Animal): ");
  3     h.makeSound();
  4 }
  5 static void print(Dog k) {
  6     System.out.print("print(Dog): ");
  7     k.makeSound();
  8 }
  9 // in main:
 10 Animal animal = new Dog();
 11 Animal other = new Cat();
 12 print(animal);
▶ print(other);
 14 print((Dog) animal);
```



`other` is also of type `Animal`: the same version is chosen.

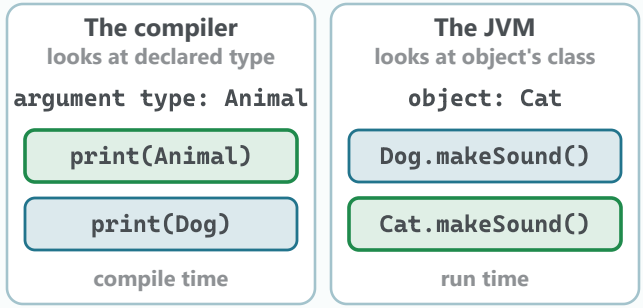
Step 4 / 7

OUTPUT

```
print(Animal): Woof woof!
print(Animal):
```

Compiler or JVM? Code Trace

```
1  static void print(Animal h) {
2      System.out.print("print(Animal): ");
▶      h.makeSound();
4  }
5  static void print(Dog k) {
6      System.out.print("print(Dog): ");
7      k.makeSound();
8  }
9  // in main:
10 Animal animal = new Dog();
11 Animal other = new Cat();
12 print(animal);
13 print(other);
14 print((Dog) animal);
```



Same line, different object: this time `Cat.makeSound()`, **Meow!**

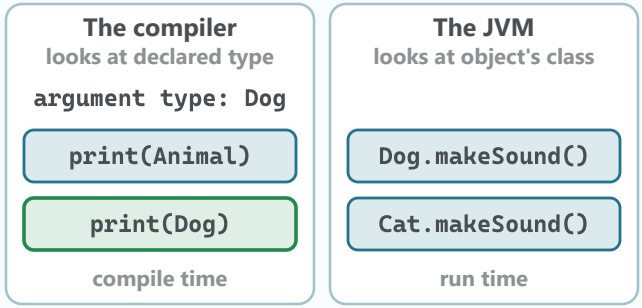
Step 5 / 7

OUTPUT

```
print(Animal): Woof woof!
print(Animal): Meow!
```

Compiler or JVM? Code Trace

```
1  static void print(Animal h) {
2      System.out.print("print(Animal): ");
3      h.makeSound();
4  }
▶ static void print(Dog k) {
▶     System.out.print("print(Dog): ");
7      k.makeSound();
8  }
9  // in main:
10 Animal animal = new Dog();
11 Animal other = new Cat();
12 print(animal);
13 print(other);
▶ print((Dog) animal);
```



The explicit cast makes the declared type `Dog`: this time the compiler picks `print(Dog)`.

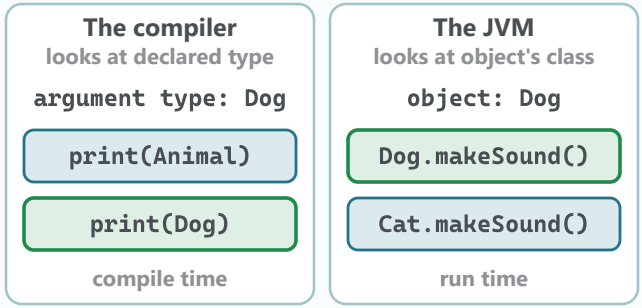
Step 6 / 7

OUTPUT

```
print(Animal): Woof woof!
print(Animal): Meow!
print(Dog):
```

Compiler or JVM? Code Trace

```
1  static void print(Animal h) {
2      System.out.print("print(Animal): ");
3      h.makeSound();
4  }
5  static void print(Dog k) {
6      System.out.print("print(Dog): ");
▶      k.makeSound();
8  }
9  // in main:
10 Animal animal = new Dog();
11 Animal other = new Cat();
12 print(animal);
13 print(other);
14 print((Dog) animal);
```



Overload resolution used the declared type **at compile time**; override selection used the actual type **at run time**.

Step 7 / 7

OUTPUT

```
print(Animal): Woof woof!
print(Animal): Meow!
print(Dog): Woof woof!
```

The Three Phases of Overload Resolution

1. No boxing

Only exact matches, primitive widening (`int` → `long`) and reference widening (`String` → `Object`) are tried.

2. Boxing

If phase 1 finds no candidate, boxing and unboxing are tried too: `double` → `Double` → `Object` .

3. Varargs

If there is still no candidate, variable-arity versions such as `int...` come into play.

- Once a phase finds at least one candidate, later phases are **skipped**; the most specific candidate of that phase wins.
- Result: given `f(long)` and `f(Integer)` , the call `f(5)` picks `f(long)` . Widening comes before boxing because boxing arrived in Java 5, and old programs had to keep their behavior.
- Narrowing (`long` → `int`) is never tried; if no phase finds a candidate: `no suitable method found for print(boolean)` .
- This order is applied at compile time; the chosen **signature** goes into the bytecode. Which class's body runs for that signature is decided at run time.

Overload Phases: Code Trace

```
1  static void f(long x)    { p("f(long)"); }
2  static void f(Integer x) { p("f(Integer)"); }
3  static void f(Object x)  { p("f(Object)"); }
4  static void f(int... x)  { p("f(int...)"); }
5  // in main (p: System.out.println):
6  byte b = 1;
▶ f(5);
8  f(b);
9  f(Integer.valueOf(5));
10 f("x");
11 f(2.5);
12 f();
13 f(1, 2);
```

Argument: 5 (int)

1. no boxing | 2. boxing | 3. varargs

`f(long)`	✓ int → long widening
`f(Integer)`	✗ needs boxing
`f(Object)`	✗ needs boxing
`f(int ... )`	

`phase 1 has one candidate: f(long)`

`5` is an `int`. In phase 1 only `long` fits, by widening; `Integer` and `Object` need boxing and are not tried in this phase.

Step 1 / 6

OUTPUT

```
f(long)
```

Overload Phases: Code Trace

```
1  static void f(long x)    { p("f(long)"); }
2  static void f(Integer x) { p("f(Integer)"); }
3  static void f(Object x)  { p("f(Object)"); }
4  static void f(int... x)  { p("f(int...)"); }
5  // in main (p: System.out.println):
6  byte b = 1;
7  f(5);
▶  f(b);
9  f(Integer.valueOf(5));
10 f("x");
11 f(2.5);
12 f();
13 f(1, 2);
```

Argument: b (byte)

1. no boxing	2. boxing	3. varargs

`f(long)`	✓ byte → long widening
`f(Integer)`	✗ Byte ≠ Integer
`f(Object)`	✗ needs boxing
`f(int ... )`	

`f(long) again`

`byte` also widens to `long`. Even boxed it would be a `Byte`, not an `Integer`.

Step 2 / 6

OUTPUT

```
f(long)
f(long)
```

Overload Phases: Code Trace

```
1  static void f(long x)    { p("f(long)"); }
2  static void f(Integer x) { p("f(Integer)"); }
3  static void f(Object x)  { p("f(Object)"); }
4  static void f(int... x)  { p("f(int...)"); }
5  // in main (p: System.out.println):
6  byte b = 1;
7  f(5);
8  f(b);
▶  f(Integer.valueOf(5));
10 f("x");
11 f(2.5);
12 f();
13 f(1, 2);
```

Argument: Integer

1. no boxing	2. boxing	3. varargs

`f(long)`	✗ needs unboxing
`f(Integer)`	✓ exact match
`f(Object)`	✓ Integer → Object
`f(int ... )`	

`two candidates: Integer is most specific`

The argument is already an `Integer` : phase 1 has two candidates. `Integer` is more specific than `Object` , so it wins.

Step 3 / 6

OUTPUT

```
f(long)
f(long)
f(Integer)
```

Overload Phases: Code Trace

```
1  static void f(long x)    { p("f(long)"); }
2  static void f(Integer x) { p("f(Integer)"); }
3  static void f(Object x)  { p("f(Object)"); }
4  static void f(int... x)  { p("f(int...)"); }
5  // in main (p: System.out.println):
6  byte b = 1;
7  f(5);
8  f(b);
9  f(Integer.valueOf(5));
▶  f("x");
11 f(2.5);
12 f();
13 f(1, 2);
```

Argument: "x" (String)

1. no boxing	2. boxing	3. varargs

- `f(long)` ✗
- `f(Integer)` ✗
- `f(Object)` ✓ String → Object
- `f(int ... )`

`reference widening: f(Object)`

A `String` fits only `Object`.

Step 4 / 6

OUTPUT

```
f(long)
f(long)
f(Integer)
f(Object)
```

Overload Phases: Code Trace

```
1  static void f(long x)    { p("f(long)"); }
2  static void f(Integer x) { p("f(Integer)"); }
3  static void f(Object x)  { p("f(Object)"); }
4  static void f(int... x)  { p("f(int...)"); }
5  // in main (p: System.out.println):
6  byte b = 1;
7  f(5);
8  f(b);
9  f(Integer.valueOf(5));
10 f("x");
▶ f(2.5);
12 f();
13 f(1, 2);
```

Argument: 2.5 (double)

1. no boxing	2. boxing	3. varargs

`f(long)`	✗ double → long narrowing
`f(Integer)`	✗ Double ≠ Integer
`f(Object)`	✓ double → Double → Object
`f(int ... )`	

`phase 1 empty → boxing in phase 2`

A `double` cannot be narrowed to `long`; phase 1 has no candidate. In phase 2 it is boxed to `Double`, which fits `Object`.

Step 5 / 6

OUTPUT

```
f(long)
f(long)
f(Integer)
f(Object)
f(Object)
```

Overload Phases: Code Trace

```
1  static void f(long x)    { p("f(long)"); }
2  static void f(Integer x) { p("f(Integer)"); }
3  static void f(Object x)  { p("f(Object)"); }
4  static void f(int... x)  { p("f(int...)"); }
5  // in main (p: System.out.println):
6  byte b = 1;
7  f(5);
8  f(b);
9  f(Integer.valueOf(5));
10 f("x");
11 f(2.5);
▶  f();
▶  f(1, 2);
```

Argument: () and (1, 2)

1. no boxing	2. boxing	3. varargs

`f(long)`	✗ parameter count
`f(Integer)`	✗ parameter count
`f(Object)`	✗ parameter count
`f(int ... )`	✓ 0 or 2 ints

`only phase 3 has a candidate`

Zero or two arguments fit only `int...`; it is tried last, in phase 3.

Step 6 / 6

OUTPUT

```
f(long)
f(long)
f(Integer)
f(Object)
f(Object)
f(int...)
f(int...)
```

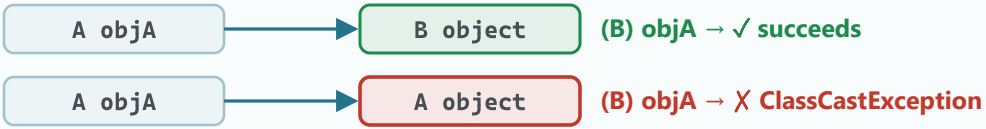
Inheritance and Polymorphism

- Inheritance is needed to build **class hierarchies**.
- Polymorphism **requires** inheritance: we need a class hierarchy.
- Inheritance **does not require** polymorphism: we can inherit just to reuse code.
- If B is a subclass of A, **a B object can act as an A object** (not the reverse).
- Assigning an A reference to a B reference needs a **narrowing conversion** (cast), written with parentheses.

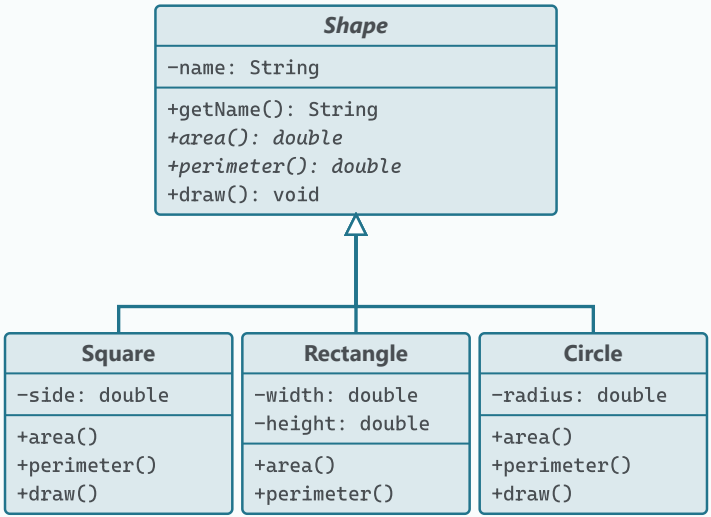
```
A objA = new B(); // B acts as an A
B objB = (B) objA; // legal, but
                   // use with care
```

The last line compiles; at run time, if `objA` does not point to a `B` object, a `ClassCastException` is thrown.

Interfaces also provide polymorphism in Java: any class that `implements` an interface can be assigned to a reference of that interface type.



Example: Shape Hierarchy




```java
abstract class Shape {
    private String name;

    public Shape(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public abstract double area();

    public abstract double perimeter();

    public void draw() {
        System.out.println(name + " drawn.");
    }
}
```

`abstract`: a method with no body; every subclass **must** implement it, and `new Shape(…)` is not allowed.

Subclasses Override the Methods

```
class Square extends Shape {
    private double side;
    public Square(double side) {
        super("Square");
        this.side = side;
    }
    @Override
    public double area() {
        return side * side;
    }
    @Override
    public double perimeter() {
        return 4 * side;
    }
    @Override
    public void draw() {
        System.out.println("Square drawn: side " + side);
    }
}
```

`Square` : the constructor passes the name up with `super("Square")` ; it overrides all three methods.

Subclasses Override the Methods

```
class Rectangle extends Shape {
    private double width;
    private double height;
    public Rectangle(double width, double height) {
        super("Rectangle");
        this.width = width;
        this.height = height;
    }
    @Override
    public double area() {
        return width * height;
    }
    @Override
    public double perimeter() {
        return 2 * (width + height);
    }
    // no draw() → Shape's is used
}
```

`Rectangle`: overrides area and perimeter but not `draw()` → `Shape.draw()` is used.

Subclasses Override the Methods

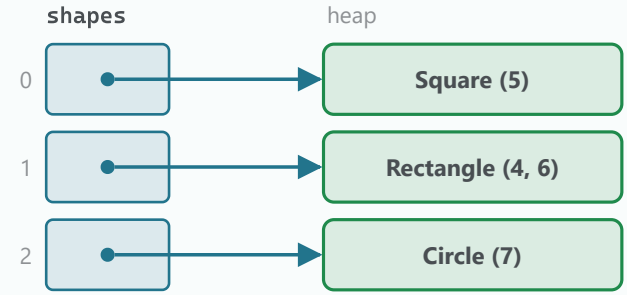
```
class Circle extends Shape {
    private double radius;
    public Circle(double radius) {
        super("Circle");
        this.radius = radius;
    }
    @Override
    public double area() {
        return Math.PI * radius * radius;
    }
    @Override
    public double perimeter() {
        return 2 * Math.PI * radius;
    }
    @Override
    public void draw() {
        System.out.println("Circle drawn: radius " + radius);
    }
}
```

`Circle` : computes with `Math.PI` ; overrides `draw()` with its own info.

Shape Array: Step-by-Step Trace

```
1  Shape[] shapes = new Shape[3];
2  shapes[0] = new Square(5);
3  shapes[1] = new Rectangle(4, 6);
▶ shapes[2] = new Circle(7);
5
6  for (Shape shape : shapes) {
7      System.out.println(shape.getName()
8          + " area: " + shape.area());
9      shape.draw();
10 }
```

OUTPUT



The array is of type `Shape`; its elements point to objects of three different subclasses.

Step 1 / 7

CALL STACK

`main()`

VARIABLES

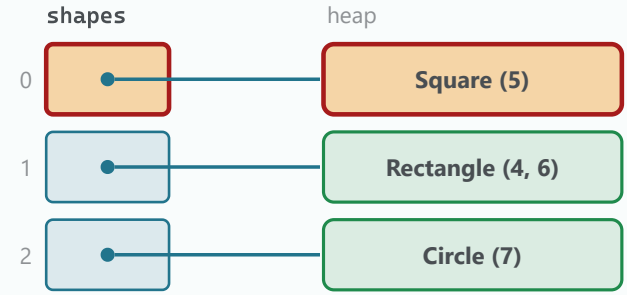
`shape = –`

Shape Array: Step-by-Step Trace

```
1  Shape[] shapes = new Shape[3];
2  shapes[0] = new Square(5);
3  shapes[1] = new Rectangle(4, 6);
4  shapes[2] = new Circle(7);
5
6  for (Shape shape : shapes) {
7      System.out.println(shape.getName()
▶          + " area: " + shape.area());
9      shape.draw();
10 }
```

OUTPUT

```
Square area: 25.0
```



`shape.area()` → `Square.area()` runs.

Step 2 / 7

CALL STACK

- **`Square.area()`**
- `main()`

VARIABLES

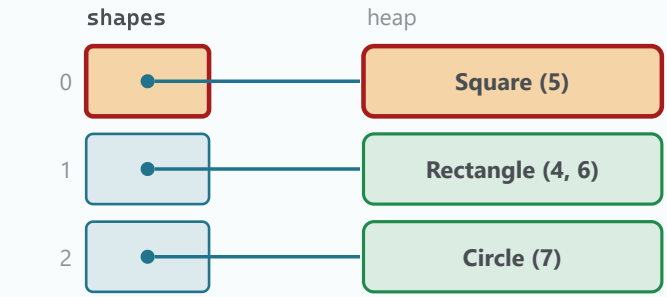
`shape` = → **Square (5)**

Shape Array: Step-by-Step Trace

```
1  Shape[] shapes = new Shape[3];
2  shapes[0] = new Square(5);
3  shapes[1] = new Rectangle(4, 6);
4  shapes[2] = new Circle(7);
5
6  for (Shape shape : shapes) {
7      System.out.println(shape.getName()
8          + " area: " + shape.area());
▶      shape.draw();
10 }
```

OUTPUT

```
Square area: 25.0
Square drawn: side 5.0
```



`draw()` overridden → `Square.draw()` runs.

Step 3 / 7

CALL STACK

- **`Square.draw()`**
- `main()`

VARIABLES

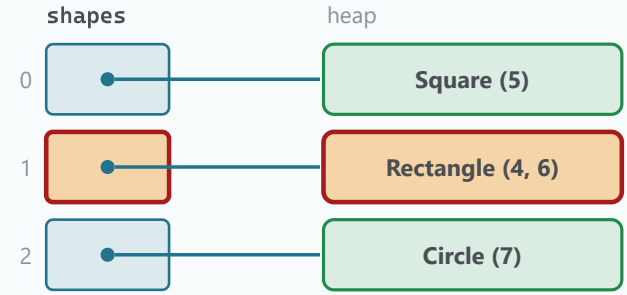
`shape` = → **Square (5)**

Shape Array: Step-by-Step Trace

```
1  Shape[] shapes = new Shape[3];
2  shapes[0] = new Square(5);
3  shapes[1] = new Rectangle(4, 6);
4  shapes[2] = new Circle(7);
5
6  for (Shape shape : shapes) {
7      System.out.println(shape.getName()
▶          + " area: " + shape.area());
9      shape.draw();
10 }
```

OUTPUT

```
Square area: 25.0
Square drawn: side 5.0
Rectangle area: 24.0
```



`shape.area()` → `Rectangle.area()` runs.

Step 4 / 7

CALL STACK

- **`Rectangle.area()`**
- `main()`

VARIABLES

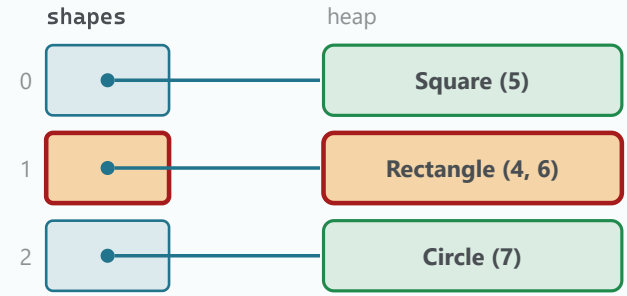
`shape` = → **Rectangle (4, 6)**

Shape Array: Step-by-Step Trace

```
1  Shape[] shapes = new Shape[3];
2  shapes[0] = new Square(5);
3  shapes[1] = new Rectangle(4, 6);
4  shapes[2] = new Circle(7);
5
6  for (Shape shape : shapes) {
7      System.out.println(shape.getName()
8          + " area: " + shape.area());
▶      shape.draw();
10 }
```

OUTPUT

```
Square area: 25.0
Square drawn: side 5.0
Rectangle area: 24.0
Rectangle drawn.
```



`Rectangle` does not override `draw()` → lookup goes up, `Shape.draw()` runs.

Step 5 / 7

CALL STACK

- **`Shape.draw()`**
- `main()`

VARIABLES

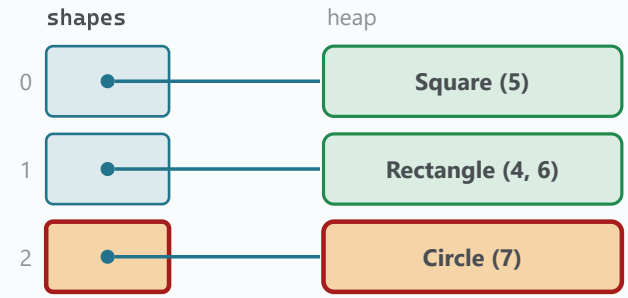
`shape` = → **Rectangle (4, 6)**

Shape Array: Step-by-Step Trace

```
1  Shape[] shapes = new Shape[3];
2  shapes[0] = new Square(5);
3  shapes[1] = new Rectangle(4, 6);
4  shapes[2] = new Circle(7);
5
6  for (Shape shape : shapes) {
7      System.out.println(shape.getName()
▶          + " area: " + shape.area());
9      shape.draw();
10 }
```

OUTPUT

```
Square area: 25.0
Square drawn: side 5.0
Rectangle area: 24.0
Rectangle drawn.
Circle area: 153.93804002589985
```



`shape.area()` → `Circle.area()` runs.

Step 6 / 7

CALL STACK

- `Circle.area()`
- `main()`

VARIABLES

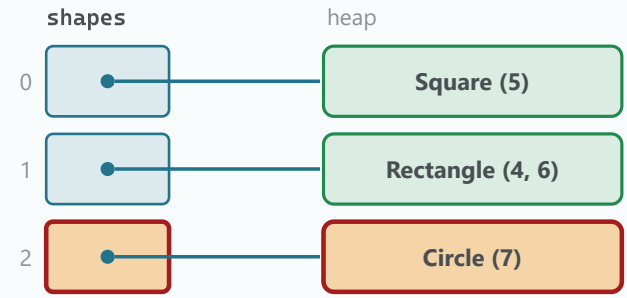
`shape = → Circle (7)`

Shape Array: Step-by-Step Trace

```
1  Shape[] shapes = new Shape[3];
2  shapes[0] = new Square(5);
3  shapes[1] = new Rectangle(4, 6);
4  shapes[2] = new Circle(7);
5
6  for (Shape shape : shapes) {
7      System.out.println(shape.getName()
8          + " area: " + shape.area());
▶      shape.draw();
10 }
```

OUTPUT

```
Square area: 25.0
Square drawn: side 5.0
Rectangle area: 24.0
Rectangle drawn.
Circle area: 153.93804002589985
Circle drawn: radius 7.0
```



`draw()` overridden → `Circle.draw()` runs.

Step 7 / 7

CALL STACK

- **`Circle.draw()`**
- `main()`

VARIABLES

`shape` = → **Circle (7)**

Why Polymorphism?

With type tests

```
static double badArea(Shape s) {
    double area;
    if (s instanceof Square) {
        area = ((Square) s).area();
    } else if (s instanceof Rectangle) {
        area = ((Rectangle) s).area();
    } else if (s instanceof Circle) {
        area = ((Circle) s).area();
    } else {
        area = 0; // new type forgotten!
    }
    return area;
}
```

Every new shape means changing this method; forget it and `badArea(new RightTriangle(3, 4))` silently returns **0.0**.

Polymorphic

```
static double totalArea(Shape[] arr) {
    double total = 0;
    for (Shape shape : arr) {
        // each object knows its area
        total += shape.area();
    }
    return total;
}
```

+ Just add a new `RightTriangle extends Shape` class; this code **never changes**.
+ { Square(5), Rectangle(4, 6), RightTriangle(3, 4) } → **55.0**
+ Shorter code, and no `ClassCastException` risk.

Principle: open for extension, closed for modification (open/closed principle).

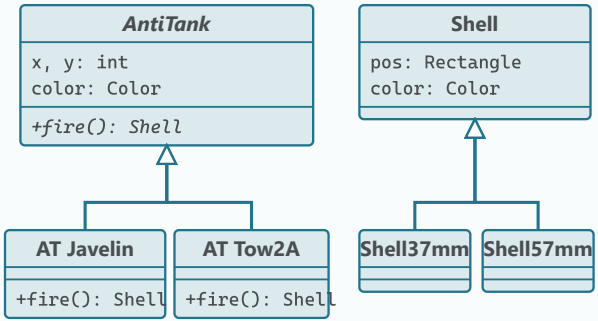
Polymorphic Parameters and Return Values

```
class Vet {
    void examine(Animal h) {      // any subclass
        System.out.print("Exam: ");
        h.makeSound();
    }
    static Animal adopt(String kind) { // hidden
        if (kind.equals("cat")) return new Cat();
        return new Dog();
    }
}
// in main:
Vet v = new Vet();
v.examine(new Cat());
Animal h = Vet.adopt("dog");
v.examine(h);
System.out.println(h.getClass().getSimpleName());
```

```
Exam: Meow!
Exam: Woof woof!
Dog
```

- **Parameter** of superclass type: the method accepts objects of every subclass, existing or yet to be written. `examine` needs no change for a new `Bird` class.
- **Return type** of superclass type: the method hides which subclass it creates. The caller relies only on the `Animal` contract; if `adopt` starts returning another subclass, client code is unaffected.
- Methods that create objects like this are called **factory methods**.
- General rule: choose the **most general** type for parameters and variables that still does the job.

Example: Anti-Tank Game



AT Javelin: `AntiTankJavelin`, AT Tow2A: `AntiTankTow2A`. `Tank` also has subclasses `TankT90MS` and `TankLeopard2A4`.

- In this Swing game you shoot falling tanks; every multiple of 10 points switches the anti-tank weapon.
- The game loop knows only the types `AntiTank` and `Tank`.

```
AntiTank weapon = new AntiTankJavelin(x, y);
...
// which shell? depends on the object (dynamic binding)
shells.add(weapon.fire());
...
if (weapon instanceof AntiTankJavelin) {
    weapon = new AntiTankTow2A(weapon.x, weapon.y);
}
```

TEST OUTPUT

```
Shell t1 = weapon.fire(); // Javelin
// → Shell37mm, height 30
weapon = new AntiTankTow2A(100, 540);
Shell t2 = weapon.fire();
// → Shell57mm, height 40
```

The same `fire()` call yields a different shell when the weapon changes; the game code needs no `if`.

`equals` and Inheritance: The Symmetry Trap

`Object` methods are dynamically bound too: `Object o = k1; System.out.println(o);` runs the object's own `toString`, and `o.equals(x)` its own `equals`.

```
class Point {
    final int x, y;
    Point(int x, int y) { this.x = x; this.y = y; }
    @Override public boolean equals(Object o) {
        if (!(o instanceof Point)) return false;
        Point n = (Point) o;
        return x == n.x && y == n.y;
    }
    @Override public int hashCode() { return 31 * x + y; }
}
class ColoredPoint extends Point {
    final String color;
    ColoredPoint(int x, int y, String color) {
        super(x, y);
        this.color = color;
    }
    @Override public boolean equals(Object o) {
        if (!(o instanceof ColoredPoint)) return false;
        ColoredPoint r = (ColoredPoint) o;
        return super.equals(r) && color.equals(r.color);
    }
}
```

```
Point p = new Point(1, 2);
ColoredPoint k =
    new ColoredPoint(1, 2, "red");
System.out.println(p.equals(k)
    + " " + k.equals(p));
```

```
true false
```

- The **receiver object** decides which `equals` runs: `p.equals(k)` runs `Point`'s, `k.equals(p)` runs `ColoredPoint`'s.
- The results differ: symmetry is broken. A `HashSet` or `contains` may answer differently depending on the order of elements.
- Fixes: compare `getClass()` in both classes; or use composition instead of inheritance (`ColoredPoint` holds a `Point` field).

Summary

Concept	Rule	Example
The reference rule	A superclass reference can point to a subclass object; the reverse won't compile	`Animal h = new Dog();`
is-a	A subclass object **is a** superclass object; not the reverse	`h instanceof Animal` → true
Upcasting	Automatic and always safe	`Object o = dog;`
Downcasting	Explicit; if the object doesn't fit, `ClassCastException`	`Dog k = (Dog) h;`
Access	The **reference type** decides which methods can be called	`h.bark()` → compile error
Dynamic binding	The **object type** decides which version runs; lookup goes bottom-up	`h.run()` → Dog version
Static binding	Fields, `static` / `private` / `final` methods and overload selection	`h.kind` → "animal"
Benefit	Code written once against the superclass works unchanged with new subclasses	`totalArea(shapes)`

SUMMARY

Related Pages

Abstract Classes and Interfaces

Defining interfaces and `implements`, polymorphism through interface references, multiple interfaces, `default` / `static` / `private` interface methods, generics.

Link to interfaces

A class can extend only one superclass but implement many interfaces: the same dynamic binding works with interface types, e.g. `SoundMaker s` instead of `Animal h`.

SUMMARY

Common Mistakes (1): Compile Errors

Code	Compiler message	Cause / fix
`Dog k = new Animal();`	*incompatible types: Animal cannot be converted to Dog*	Not every animal is a dog; the reverse assignment is illegal.
`Dog k = animal;`	*incompatible types: Animal cannot be converted to Dog*	Even if the object is a dog, an explicit cast is needed: `(Dog) animal`.
`animal.bark();`	*cannot find symbol … variable animal of type Animal*	The reference type has no such method; downcast first.
`(Dog) h.bark();`	*not a statement*	Missing parentheses: `((Dog) h).bark();`
`(Cat) dog`	*incompatible types: Dog cannot be converted to Cat*	There is no cast between sibling classes.
`@Override void makesound()`	*method does not override or implement a method from a supertype*	Name misspelled (case matters).
`void makeSound()` (super: `public`)	*attempting to assign weaker access privileges; was public*	An overriding method cannot reduce access.

SUMMARY

Common Mistakes (2): Run Time and Logic

Unchecked downcast

Writing `(Dog) h` and forgetting that `h` may point to a `Cat` → `ClassCastException`. Test with `instanceof` first.

Thinking "a cast changes the object"

`(Animal) dog` does not turn the dog into an animal; `makeSound()` still prints "Woof woof!".

Thinking fields are overridden too

`h.kind` is read by the reference type ("animal"). Fix: make fields `private` and override a getter instead.

Confusing overloading with overriding

Writing `equals(Person p)` in a subclass does not override `equals(Object)`, it overloads it; collections never call this method. `@Override` catches this mistake.

`==` to compare objects

`==` compares references; for content, use the overridden `equals`.

`instanceof` chains everywhere

They break as subclasses are added. Move the behavior into an overridden method.

SUMMARY

Common Mistakes (3)

Calling an overridable method from a constructor

The subclass method runs on an object whose fields are not yet set: `null` output or `NullPointerException`.

Relying on array covariance

`Animal[] d = new Dog[2]; d[1] = new Cat();` compiles, but at run time: `ArrayStoreException: Cat`.

Ambiguous call with null

`show(null)`, with two unrelated overloads: `reference to show is ambiguous`. Resolve it with an explicit cast.

Thinking boxing comes first

With `f(long)` and `f(Integer)`, `f(5)` → `f(long)`: widening is in phase 1.

Extending equals one-way in a subclass

`p.equals(k)` true, `k.equals(p)` false: symmetry breaks, collections behave inconsistently.

Treating a generic list like an array

`List<Animal> l = new ArrayList<Dog>();` → `incompatible types`. Generics are not covariant.

Sample Problem 1: What Is the Output?

```
class A {
    void print() {
        System.out.println("A.print");
    }

    void call() {
        print(); // this.print()
    }
}

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

// in main:
A a = new B();
a.call();
new A().call();
```

`call()` is defined only in `A` and calls `print()`. `B` overrides only `print()`. What does the program print?

▶ **Answer**

SUMMARY

Sample Problem 2: Does It Compile? Does It Run?

```
Animal h = new Cat();
Cat k1 = h;                 // (1)
Cat k2 = (Cat) h;           // (2)
Dog k3 = (Dog) h;           // (3)
h.scratch();                // (4)
((Cat) h).scratch();        // (5)
Object o = new Dog();
o.makeSound();              // (6)
((Animal) o).makeSound(); // (7)
```

For each line decide: **compile error**, **run-time error** or **fine** (with output).

▶ **Answer**

Sample Problem 3: Overloading + Overriding

```
class Parent {
    void f(Object o) {
        System.out.println("Parent.f(Object)");
    }

    void f(String s) {
        System.out.println("Parent.f(String)");
    }
}

class Child extends Parent {
    @Override
    void f(Object o) {
        System.out.println("Child.f(Object)");
    }
}

// in main:
Parent u = new Child();
Object o = "hello";
u.f(o);
u.f("hello");
```

`Parent` has two overloaded `f` methods; `Child` overrides only `f(Object)`. What do the two calls print?

▶ **Answer**

SUMMARY

Sample Problem 4: Construction, Phases, Arrays

```
class Parent {
    Parent() { show(); }
    void show() { System.out.println("Parent"); }
}
class Child extends Parent {
    private int[] arr = {1, 2, 3};
    @Override void show() {
        System.out.println(arr == null
            ? "no arr" : "length " + arr.length);
    }
}
```

1. `new Child();` and then `new Child().show();` print what?
2. Given `f(long)`, `f(Integer)`, `f(Object)` and `f(int...)`, which versions do `short s = 3; f(s);`, `f('A');` and `f(Long.valueOf(4));` choose?
3. `Object[] d = new String[1]; d[0] = 1;` : compiles? runs?

▶ **Show answers**

SUMMARY

Exercises (1)

1. Write `Employee` (`salary()` = base salary), `Manager` (base × 1.5) and `Intern` (fixed 5000). For `Employee[]` = { Ayşe 20000, Manager Mehmet 30000, Intern Can }, what is the total salary?
2. `X` : `name()` → "X", `print()` → `println(name())` . `Y extends X` : `name()` → "Y". `Z extends Y` : `print()` → `println("Z:" + name())` . What is printed if `print()` is called on each of { new X(), new Y(), new Z() }?
3. `Object o = new Dog();` followed by `o.makeSound();` : why doesn't it compile? How do you fix it, and what does it print then?

▶ **Answers / hints**

Exercises (2)

4. Add `RightTriangle(a, b)` to the shape hierarchy (area = a·b/2, perimeter = a + b + hypotenuse). For `RightTriangle(3, 4)`, what are the area, perimeter and `draw()` outputs? What changes in the shape loop?
5. Rewrite the `instanceof` chain loop (Dog → bark, Cat → scratch, otherwise → "No special behavior") without any `instanceof`.
6. Given overloads `print(Animal)` and `print(Dog)`, does `Object o = new Dog(); print(o);` compile?

▶ **Answers / hints**

References

- Oracle, *The Java Tutorials*: Polymorphism; Overriding and Hiding Methods; Inheritance.
- J. Gosling et al., *The Java Language Specification*, Java SE 21 Edition (5.5 Casting Contexts, 8.4.8 Inheritance, Overriding, and Hiding, 15.12 Method Invocation Expressions).
- P. Deitel and H. Deitel, *Java: How to Program*, 11th ed., Pearson (Chapter 10: Object-Oriented Programming: Polymorphism and Interfaces).
- K. Sierra and B. Bates, *Head First Java*, 3rd ed., O'Reilly (Inheritance and Polymorphism).
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12th ed., Pearson (Chapter 5: Inheritance).
- A. Downey and C. Mayfield, *Think Java*, 2nd ed., O'Reilly.

The End

Chapter 4.4: Polymorphism

Previous: Chapter 4.3: Inheritance · Next: Chapter 4.5: Abstract Classes and Interfaces