Polymorphism in Java

Goals While Writing Computer Code

Correctness and Robustness

- Works as expected on normal inputs
- Capable of handling or prevents unexpected inputs

Flexibility and Adaptability

- Software needs to be able to evolve over time in response to changing conditions in its environment

Reusability

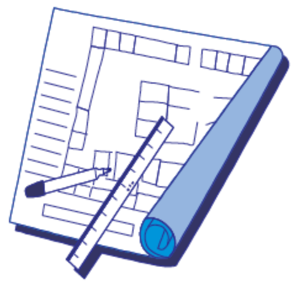
- The same code should be usable as a component of different systems in various applications

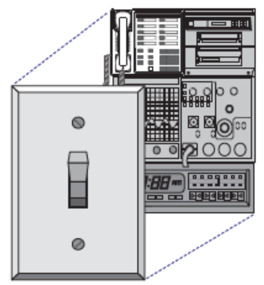
Efficiency

Readability, Ease of Maintenance

…

Modularity


Abstraction


Encapsulation

Object-Oriented Design Principles

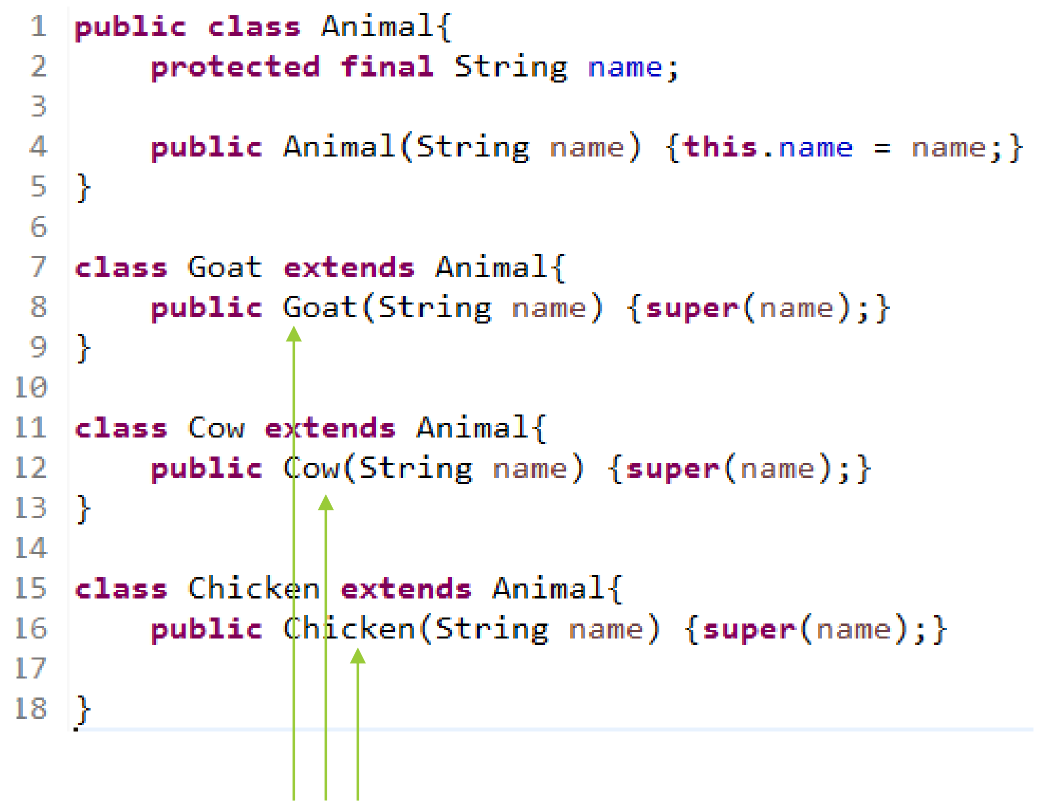
Scenario: Animal Farm

Trying to implement a farm simulation video game

Have some types of farm animals: goats, cows, and chickens

- All share some features, e.g., all our animals have a name
- All may have something special: e.g., amount of wool per season, milk per day, eggs per week (not shown)

Inheritance makes sense



```java
public class Animal{
    protected final String name;

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

class Goat extends Animal{
    public Goat(String name) {super(name);}
}

class Cow extends Animal{
    public Cow(String name) {super(name);}
}

class Chicken extends Animal{
    public Chicken(String name) {super(name);}

}
```

Reminder: Constructors are never inherited in Java

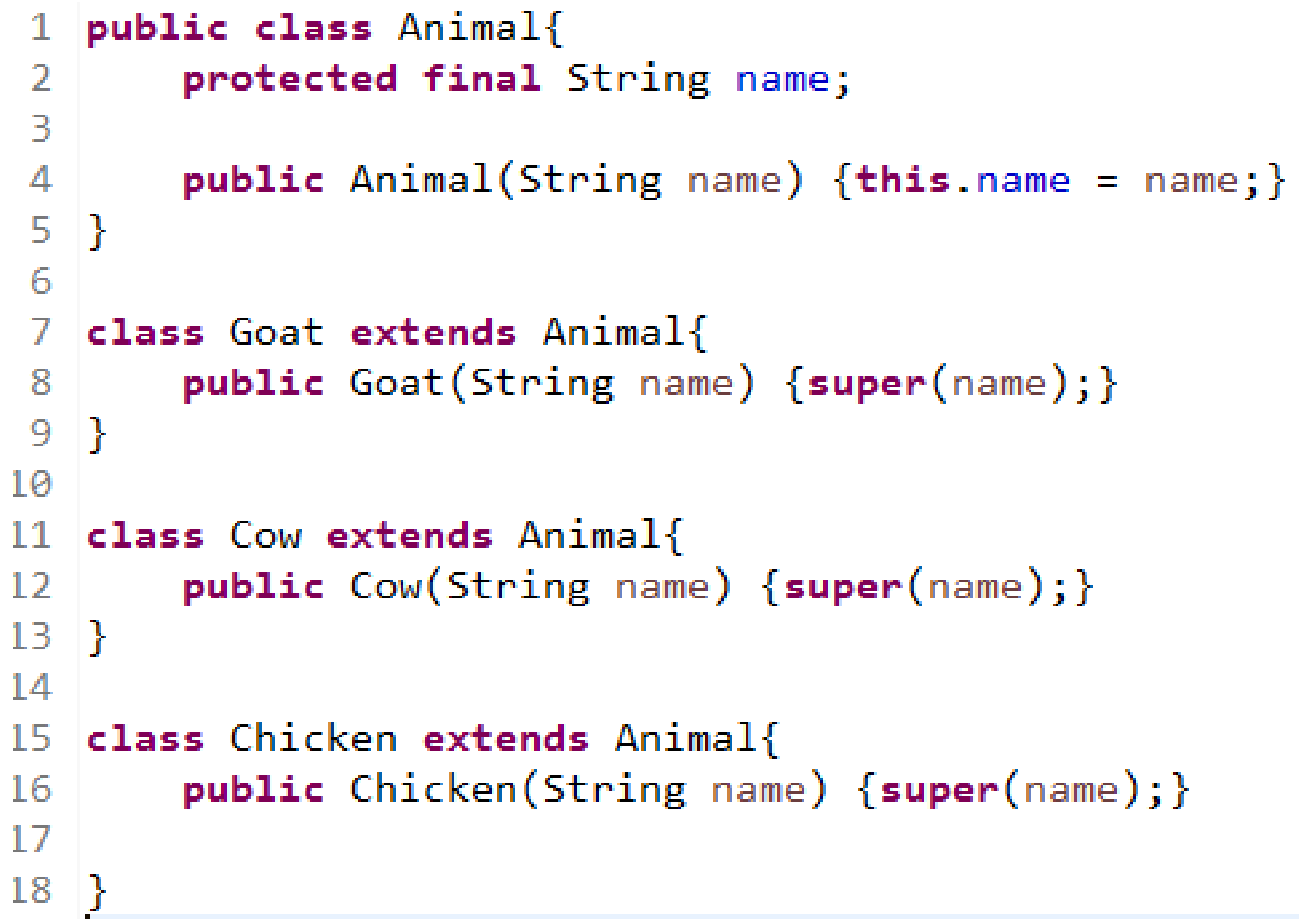
Scenario: Animal Farm

Trying to implement a farm simulation video game

Have some **types** of farm animals: goats, cows, and chickens

- All share some features, e.g., all our animals have a name
- All may have something special: e.g., amount of wool per season, milk per day, eggs per week (not shown)
- Special behaviours: animals need to make various noises
  - Alternatively: animal objects need draw themselves on the screen

How do we implement it?



```java
public class Animal{
    protected final String name;

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

class Goat extends Animal{
    public Goat(String name) {super(name);}
}

class Cow extends Animal{
    public Cow(String name) {super(name);}
}

class Chicken extends Animal{
    public Chicken(String name) {super(name);}

}
```

Animal Farm: pass type as a parameter

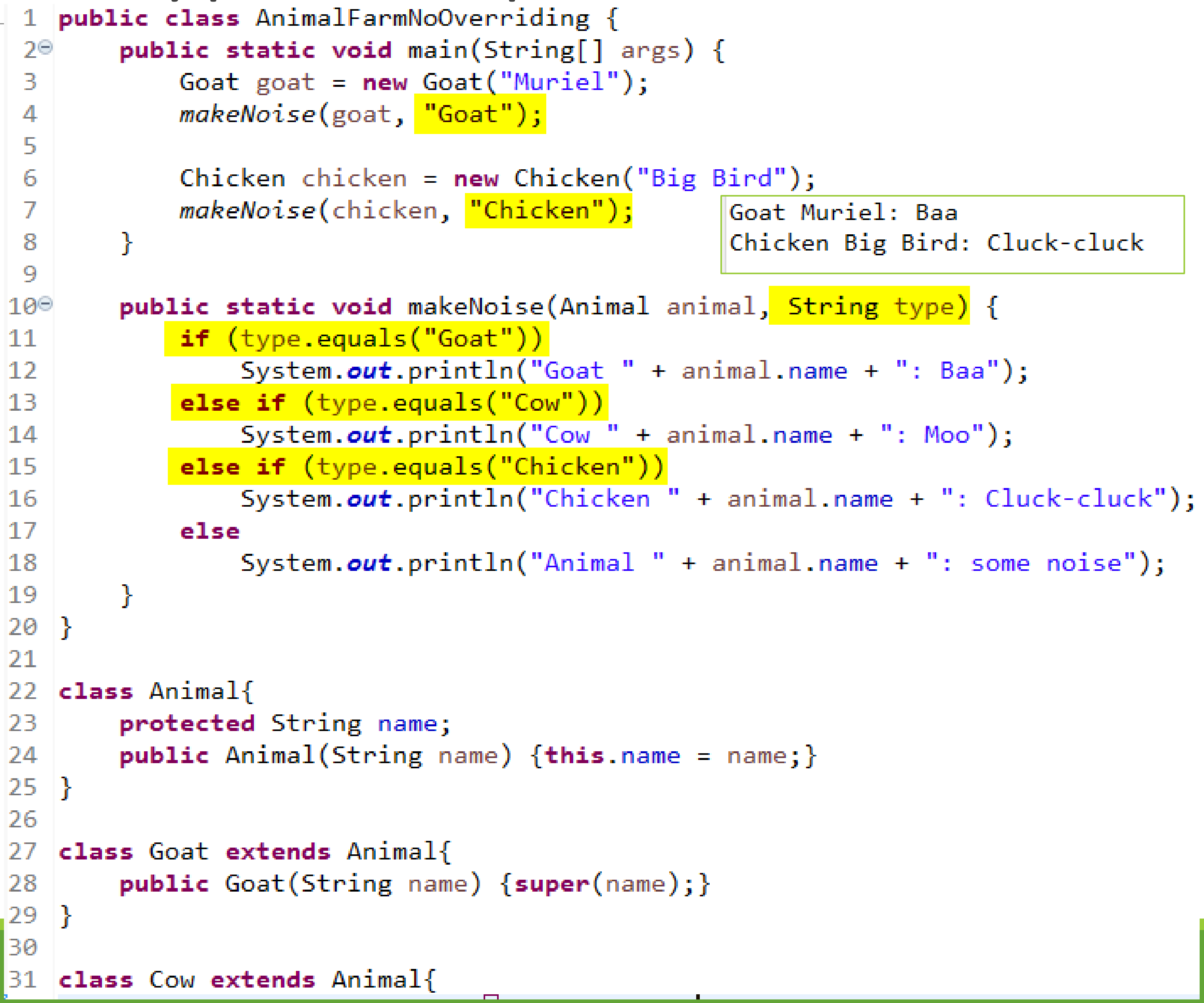
Trying to implement a farm simulation video game

Have some types of farm animals: goats, cows, and chickens

- All share some features, e.g., all our animals have a name
- All may have something special: e.g., amount of wool per season, milk per day, eggs per week (not shown)
- Special behaviours: animals need to make various noises

How do we implement it?

- Pass type as a parameter (here: as a String)
- Works correctly
- Problem 1: can make a mistake when specifying the type
- Problem 2: code is disconnected from the animal classes
- Problem 3: **makeNoise** is long and would become hard to manage



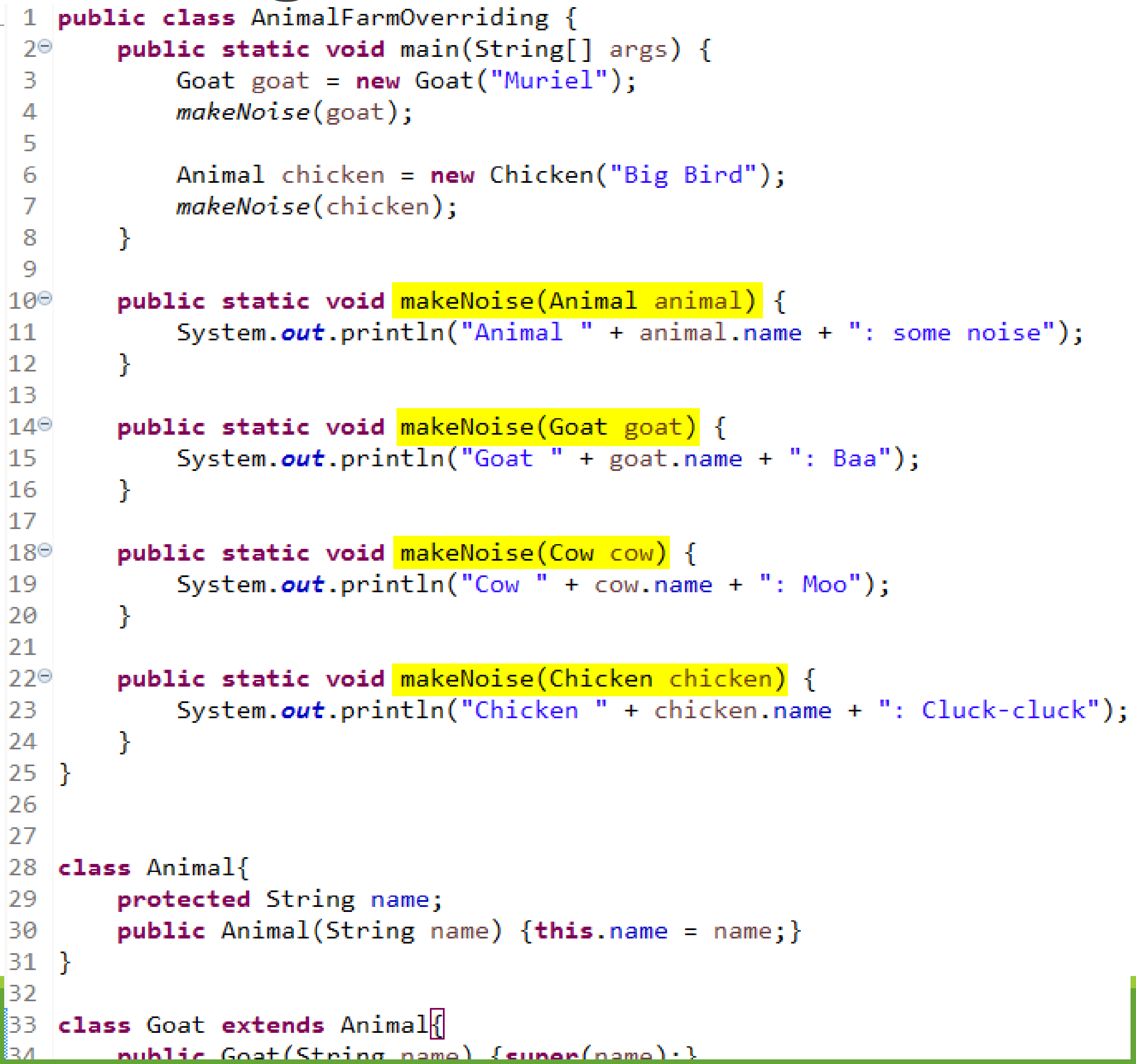
```java
public class AnimalFarmNoOverriding {
    public static void main(String[] args) {
        Goat goat = new Goat("Muriel");
        makeNoise(goat, "Goat");

        Chicken chicken = new Chicken("Big Bird");
        makeNoise(chicken, "Chicken");
    }

    public static void makeNoise(Animal animal, String type) {
        if (type.equals("Goat"))
            System.out.println("Goat " + animal.name + ": Baa");
        else if (type.equals("Cow"))
            System.out.println("Cow " + animal.name + ": Moo");
        else if (type.equals("Chicken"))
            System.out.println("Chicken " + animal.name + ": Cluck-cluck");
        else
            System.out.println("Animal " + animal.name + ": some noise");
    }
}

class Animal{
    protected String name;
    public Animal(String name) {this.name = name;}
}

class Goat extends Animal{
    public Goat(String name) {super(name);}
}

class Cow extends Animal{
```

```
Goat Muriel: Baa
Chicken Big Bird: Cluck-cluck
```

Animal Farm: overloading

How do we implement it? Another way

- Java is a typed language: use *overloading*
- Problem 1 fixed: types are now automatically associated with the variables
- Problem 2: code is disconnected from the animal classes
- Problem 3 fixed: code is separated into smaller pieces

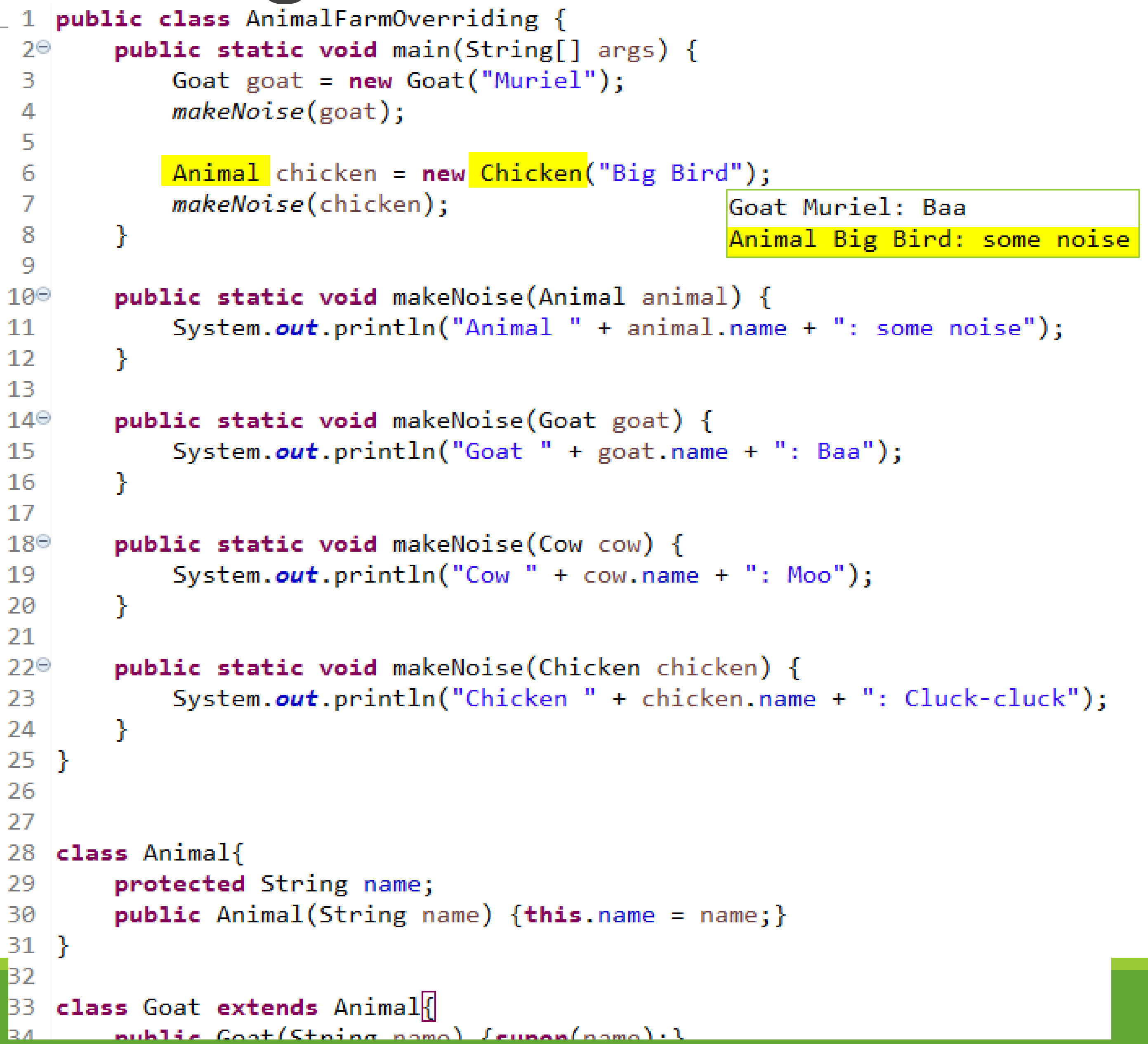


```
public class AnimalFarmOverriding {
    public static void main(String[] args) {
        Goat goat = new Goat("Muriel");
        makeNoise(goat);

        Animal chicken = new Chicken("Big Bird");
        makeNoise(chicken);
    }

    public static void makeNoise(Animal animal) {
        System.out.println("Animal " + animal.name + ": some noise");
    }

    public static void makeNoise(Goat goat) {
        System.out.println("Goat " + goat.name + ": Baa");
    }

    public static void makeNoise(Cow cow) {
        System.out.println("Cow " + cow.name + ": Moo");
    }

    public static void makeNoise(Chicken chicken) {
        System.out.println("Chicken " + chicken.name + ": Cluck-cluck");
    }
}


class Animal{
    protected String name;
    public Animal(String name) {this.name = name;}
}

class Goat extends Animal{
    public Goat(String name) {super(name);}
```

Animal Farm: overloading

How do we implement it? Another way

- Java is a typed language: use *overloading*
- Problem 1 fixed: types are now automatically associated with the variables
- Problem 2: code is disconnected from the animal classes
- Problem 3 fixed: code is separated into smaller pieces
- Problem 4 new: unexpected printing behaviour?



```java
public class AnimalFarmOverriding {
    public static void main(String[] args) {
        Goat goat = new Goat("Muriel");
        makeNoise(goat);

        Animal chicken = new Chicken("Big Bird");
        makeNoise(chicken);
    }

    public static void makeNoise(Animal animal) {
        System.out.println("Animal " + animal.name + ": some noise");
    }

    public static void makeNoise(Goat goat) {
        System.out.println("Goat " + goat.name + ": Baa");
    }

    public static void makeNoise(Cow cow) {
        System.out.println("Cow " + cow.name + ": Moo");
    }

    public static void makeNoise(Chicken chicken) {
        System.out.println("Chicken " + chicken.name + ": Cluck-cluck");
    }
}


class Animal{
    protected String name;
    public Animal(String name) {this.name = name;}
}

class Goat extends Animal{
    public Goat(String name) {super(name);}
```

```
Goat Muriel: Baa
Animal Big Bird: some noise
```

Animal Farm: can we do better?

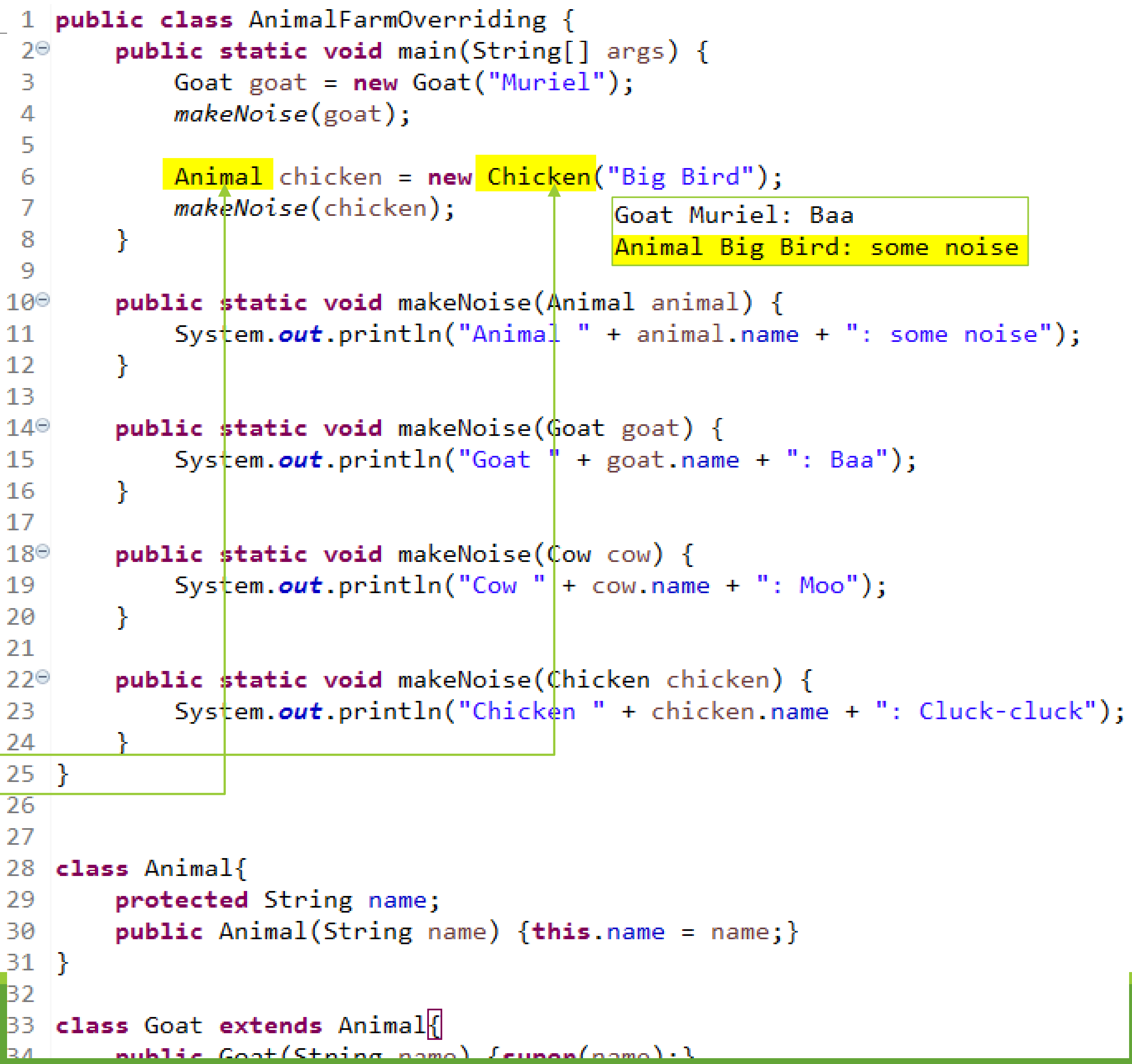
Trying to implement a farm simulation video game

Have some types of farm animals: goats, cows, and chickens

- All share some features, e.g., all our animals have a name
- All may have something special: e.g., amount of wool per season, milk per day, eggs per week (not shown)
- Special behaviours: animals need to make various noises

How do we *wish* to implement it?

- It would be great to put the printing code into the objects
- It would be great to base the behaviour on the actual object type, and not on the type of its reference



```java
public class AnimalFarmOverriding {
    public static void main(String[] args) {
        Goat goat = new Goat("Muriel");
        makeNoise(goat);

        Animal chicken = new Chicken("Big Bird");
        makeNoise(chicken);
    }

    public static void makeNoise(Animal animal) {
        System.out.println("Animal " + animal.name + ": some noise");
    }

    public static void makeNoise(Goat goat) {
        System.out.println("Goat " + goat.name + ": Baa");
    }

    public static void makeNoise(Cow cow) {
        System.out.println("Cow " + cow.name + ": Moo");
    }

    public static void makeNoise(Chicken chicken) {
        System.out.println("Chicken " + chicken.name + ": Cluck-cluck");
    }
}


class Animal{
    protected String name;
    public Animal(String name) {this.name = name;}
}

class Goat extends Animal{
    public Goat(String name) {super(name);}
```

```
Goat Muriel: Baa
Animal Big Bird: some noise
```

Animal Farm: overriding

Trying to implement a farm simulation video game

Have some types of farm animals: goats, cows, and chickens

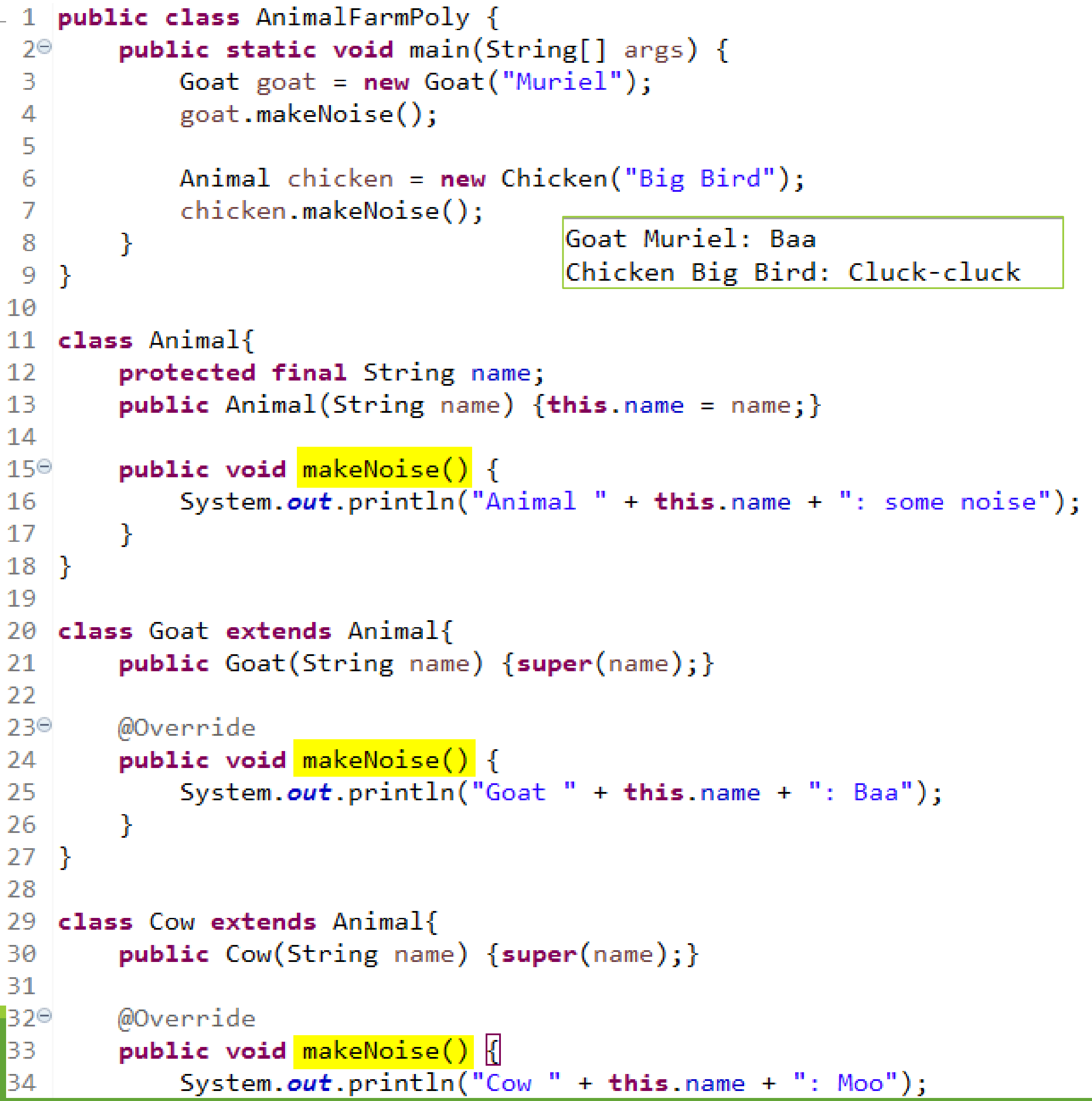
- All share some features, e.g., all our animals have a name
- All may have something special: e.g., amount of wool per season, milk per day, eggs per week (not shown)
- Special behaviours: animals need to make various noises

How do we *wish* to implement it?

- It would be great to put the printing code into the objects
- It would be great to base the behaviour on the actual object type, and not on the type of its reference

Thus, add a method **makeNoise()** and override it in every child class

Works as expected!



```java
public class AnimalFarmPoly {
    public static void main(String[] args) {
        Goat goat = new Goat("Muriel");
        goat.makeNoise();

        Animal chicken = new Chicken("Big Bird");
        chicken.makeNoise();
    }
}

class Animal{
    protected final String name;
    public Animal(String name) {this.name = name;}

    public void makeNoise() {
        System.out.println("Animal " + this.name + ": some noise");
    }
}

class Goat extends Animal{
    public Goat(String name) {super(name);}

    @Override
    public void makeNoise() {
        System.out.println("Goat " + this.name + ": Baa");
    }
}

class Cow extends Animal{
    public Cow(String name) {super(name);}

    @Override
    public void makeNoise() {
        System.out.println("Cow " + this.name + ": Moo");
```

```
Goat Muriel: Baa
Chicken Big Bird: Cluck-cluck
```

Animal Farm: using lists of animals

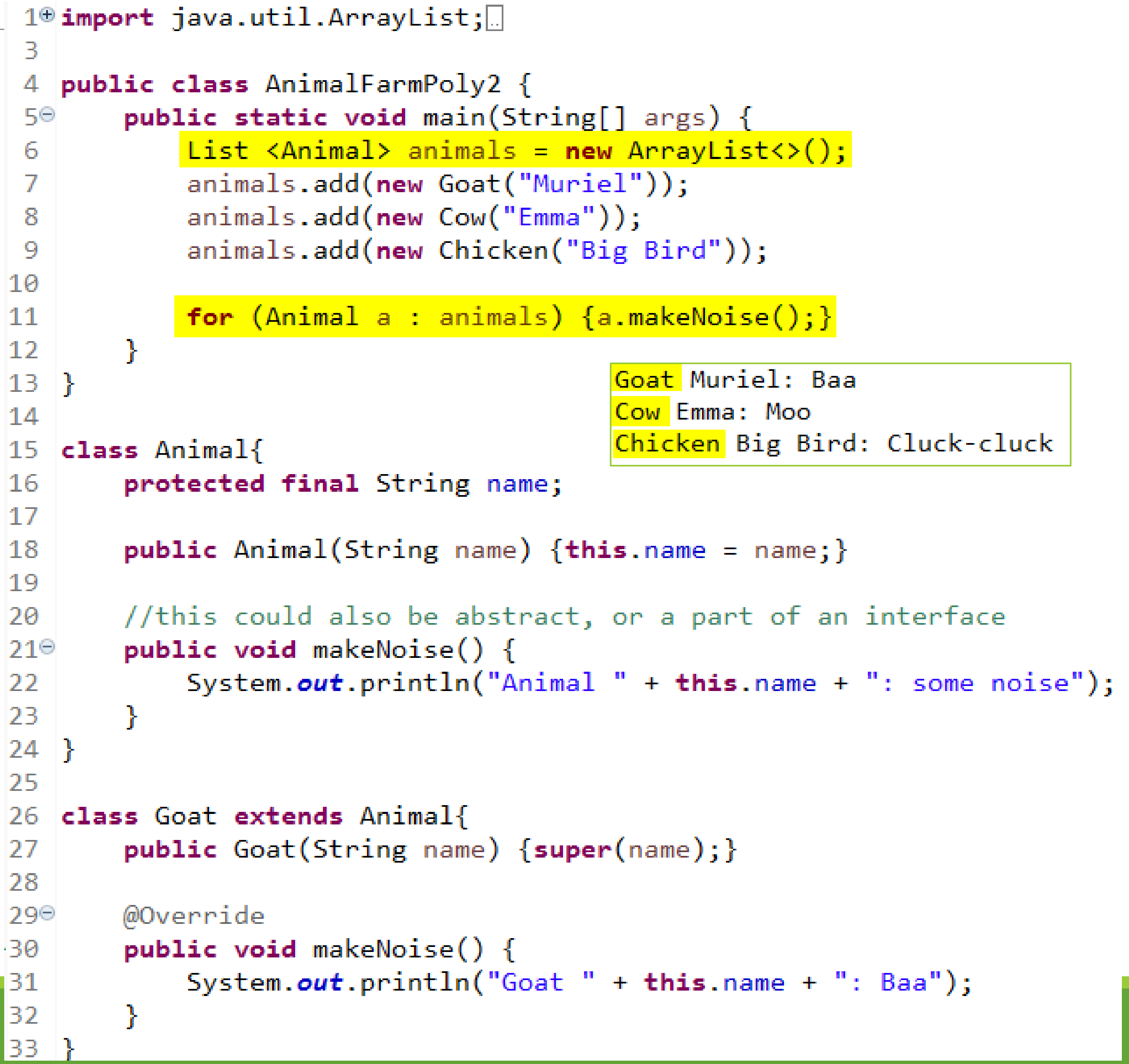
Our code is modular

Uses encapsulation

Methods are polymorphic

Let's add a collection

- Collection of **`Animal`** type
- What happens if we run this?
- Methods still work as expected!



```java
import java.util.ArrayList;

public class AnimalFarmPoly2 {
    public static void main(String[] args) {
        List <Animal> animals = new ArrayList<>();
        animals.add(new Goat("Muriel"));
        animals.add(new Cow("Emma"));
        animals.add(new Chicken("Big Bird"));

        for (Animal a : animals) {a.makeNoise();}
    }
}

class Animal{
    protected final String name;

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

    //this could also be abstract, or a part of an interface
    public void makeNoise() {
        System.out.println("Animal " + this.name + ": some noise");
    }
}

class Goat extends Animal{
    public Goat(String name) {super(name);}

    @Override
    public void makeNoise() {
        System.out.println("Goat " + this.name + ": Baa");
    }
}
```

```
Goat Muriel: Baa
Cow Emma: Moo
Chicken Big Bird: Cluck-cluck
```

Polymorphism: what

The same operation name among objects in an inheritance hierarchy may behave differently

Example: Classes `Goat`, `Cow`, and `Chicken` – are all subclasses of `Animal`

```
Animal goat = new Goat();
Animal cow = new Cow();
Animal chicken = new Chicken();
```

Calling `goat.makeNoise()`, `cow.makeNoise()`, and `chicken.makeNoise()` will call different [appropriate] methods

Polymorphism: how

Dynamic Dispatch is used with overridden methods

JVM determines which class the object is an instance of at run-time

Then calls a proper overridden method

[vs. overloaded methods: parameter types are checked at compile-time (static dispatch)]

Polymorphism requires *late binding* of the method name to the method definition

- Late binding means that the method definition is determined at run-time

non-static method

```
obj.toString()
```

run-time type of
the instance **`obj`**

Declared vs. Run-time type

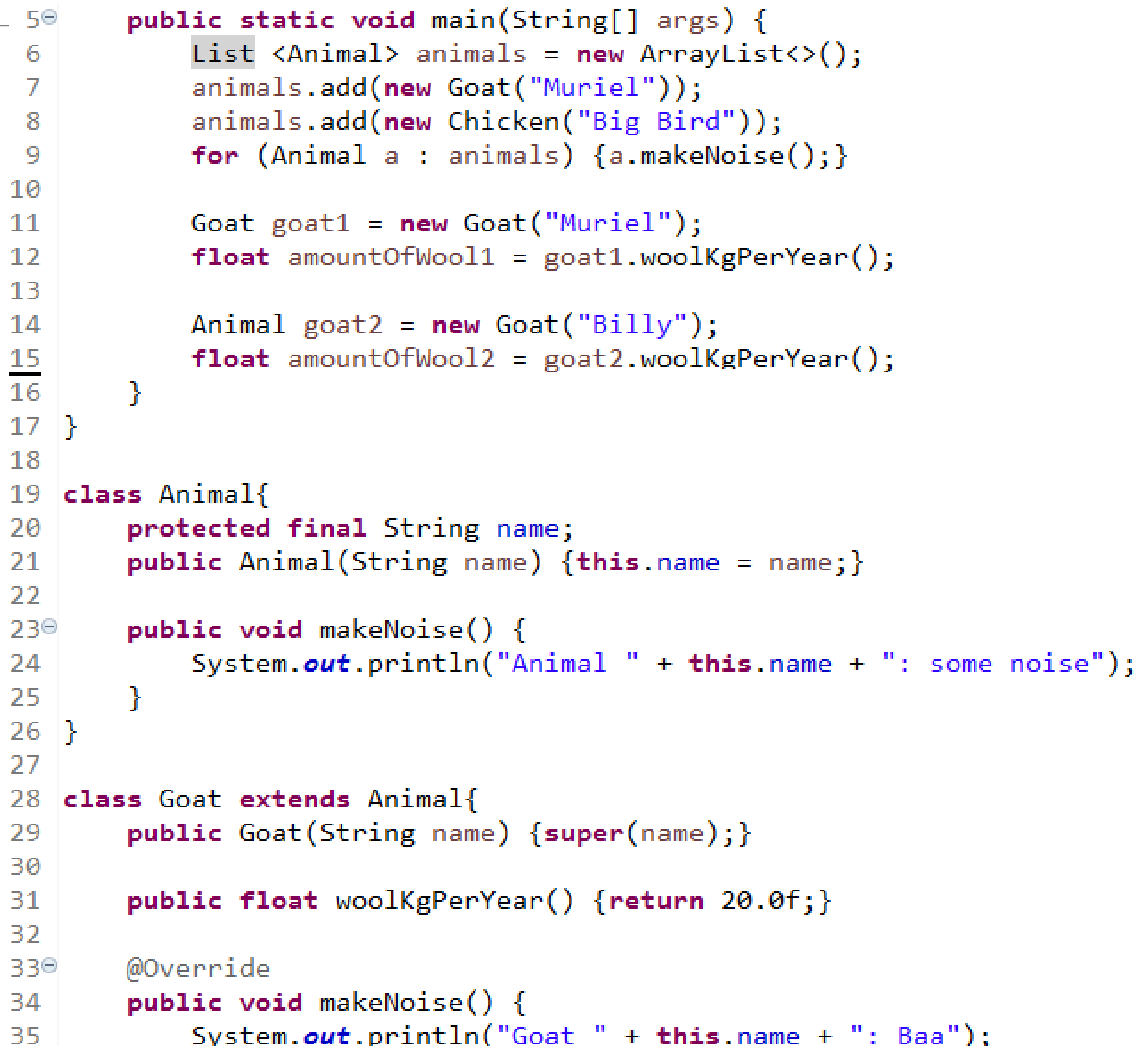
```
Animal muriel = new Goat();
```

declared
type

run-time or actual
type

Declared types matter!

```
List <Animal> animals = new ArrayList<>();

animals.add(new Goat("Muriel"));

animals.add(new Chicken("Big Bird"));

for (Animal a : animals) {a.makeNoise();}



Goat goat1 = new Goat("Muriel");

float amountOfWool1 = goat1.woolKgPerYear();


Animal goat2 = new Goat("Billy");

float amountOfWool2 = goat2.woolKgPerYear();
```



```
    public static void main(String[] args) {
        List <Animal> animals = new ArrayList<>();
        animals.add(new Goat("Muriel"));
        animals.add(new Chicken("Big Bird"));
        for (Animal a : animals) {a.makeNoise();}

        Goat goat1 = new Goat("Muriel");
        float amountOfWool1 = goat1.woolKgPerYear();

        Animal goat2 = new Goat("Billy");
        float amountOfWool2 = goat2.woolKgPerYear();
    }
}

class Animal{
    protected final String name;
    public Animal(String name) {this.name = name;}

    public void makeNoise() {
        System.out.println("Animal " + this.name + ": some noise");
    }
}

class Goat extends Animal{
    public Goat(String name) {super(name);}

    public float woolKgPerYear() {return 20.0f;}

    @Override
    public void makeNoise() {
        System.out.println("Goat " + this.name + ": Baa");
```

Declared types matter!

```
List <Animal> animals = new ArrayList<>();
animals.add(new Goat("Muriel"));
animals.add(new Chicken("Big Bird"));
for (Animal a : animals) {a.makeNoise();}
```

All `Animal` classes have `makeNoise()`
A proper version is going to be used every time due to polymorphism
`List` can store any `Animal`

```
Goat goat1 = new Goat("Muriel");
float amountOfWool1 = goat1.woolKgPerYear();
```

Only `Goat` objects have `woolKgPerYear()`

```
Animal goat2 = new Goat("Billy");
float amountOfWool2 = goat2.~~woolKgPerYear~~();
```

`goat2.woolKgPerYear()`
will not compile
Even though the object itself is of that type…

Declared vs. Run-time type

```
Animal muriel = new Goat();
```

declared type
determines what
methods *can* be used
Here, `makeNoise()`

run-time or actual type
determines *what definition* is
used when the method is called
(via Dynamic Dispatch)
Here, the **`Goat`** version

Overloading example: can we fix it?

Yes

Since there is no dynamic dispatch (methods are **static**), we can check the types ourselves:

```
public static void makeNoise(Animal animal) {
    if (animal instanceof Goat) {
        System.out.println("Goat " + animal.name + ": Baa");
    }
    else if (animal instanceof Cow) {
        System.out.println("Cow " + animal.name + ": Moo");
    }
    else if (animal instanceof Chicken) {
        System.out.println("Chicken " + animal.name + ": Cluck-cluck");
    }
    else
        System.out.println("Animal " + animal.name + ": some noise");
}
```

After that, the following code will work as we'd expected

```
Animal chicken = new Chicken("Big Bird");
makeNoise(chicken);
```

Still, more difficult to manage such code

Polymorphism

Makes the programming code easier to manage (behaviours are attached to data)

Makes the code more flexible and powerful (correct method version is always called)

Eliminates certain errors (as one can rely on strong typing)