Tutorial #6

SFWR ENG / COMP SCI 2S03

Classes, Objects, Inheritance and Overriding

Dillon Dixon

Week of October 14, 2013

What is a class?

- In object-oriented programming, a class is a *definition* of a distinct type.

- Classes encapsulate properties (variables) and actions (methods) of *“real world”* objects.

Example

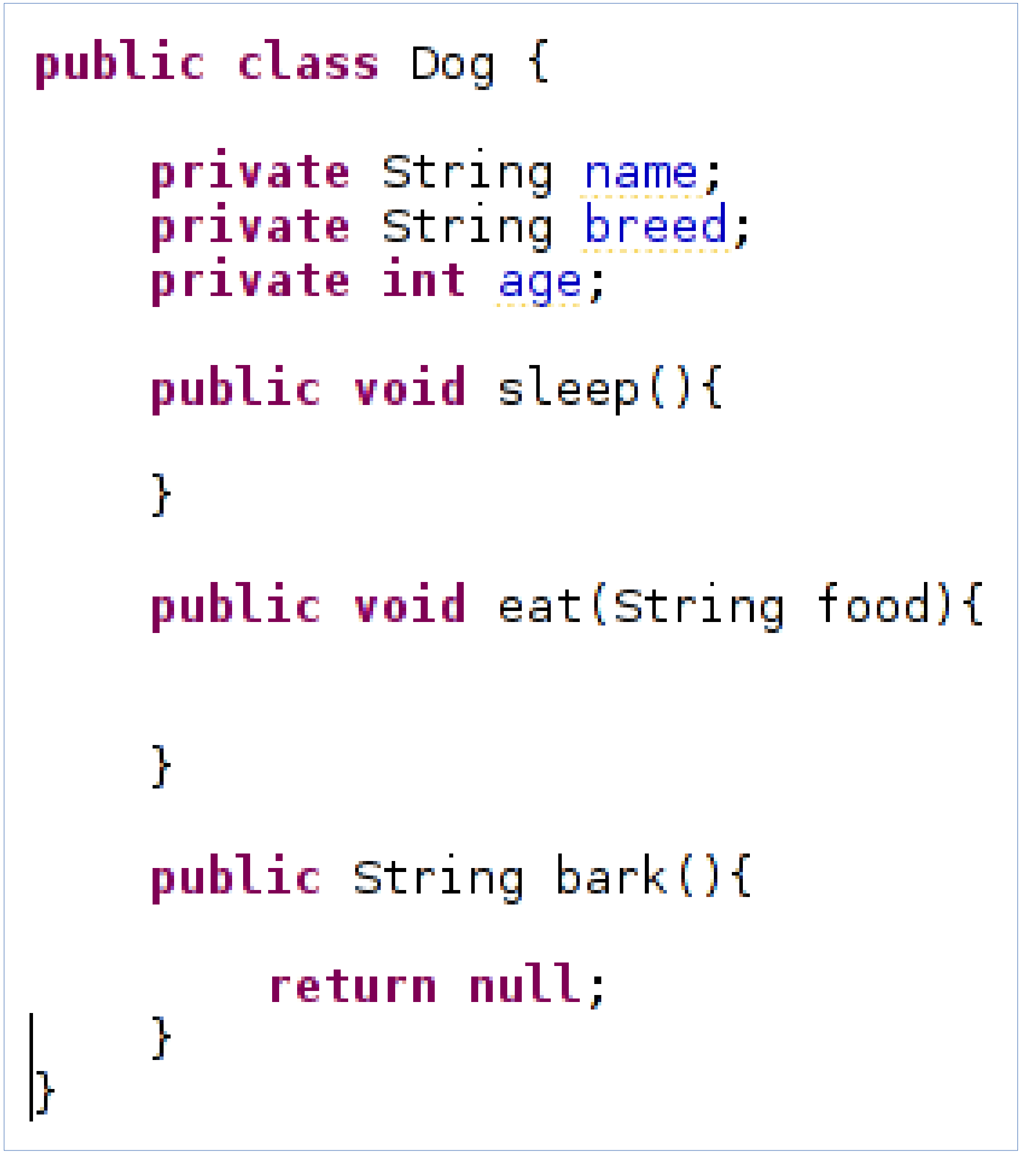
- Assume we want to make a class for dogs.

Properties:	**Variables:**
· Name	· String name
· Breed	· String breed
· Age	· int age

Actions:	**Methods:**
· Sleep	· void sleep()
· Eat	· void eat(String food)
· Bark	· String bark()

Example

- This is our equivalent Java class *so far.*
- None of the methods are implemented.



```java
public class Dog {

    private String name;
    private String breed;
    private int age;

    public void sleep(){

    }

    public void eat(String food){


    }

    public String bark(){

        return null;
    }
}
```

What is an object?

- An object is an *instance* of a class.
- Objects *define* values for the properties of a class, and sometimes even the actions as well.
- An object of a class is built using a *constructor.*

Example

Dog is the class while those at the bottom are *instantiations* or objects of that class.

Dog

Name: Jeff
Breed: German Shepherd
Age: 5

Name: Derp
Breed: Chihuahua
Age: 2

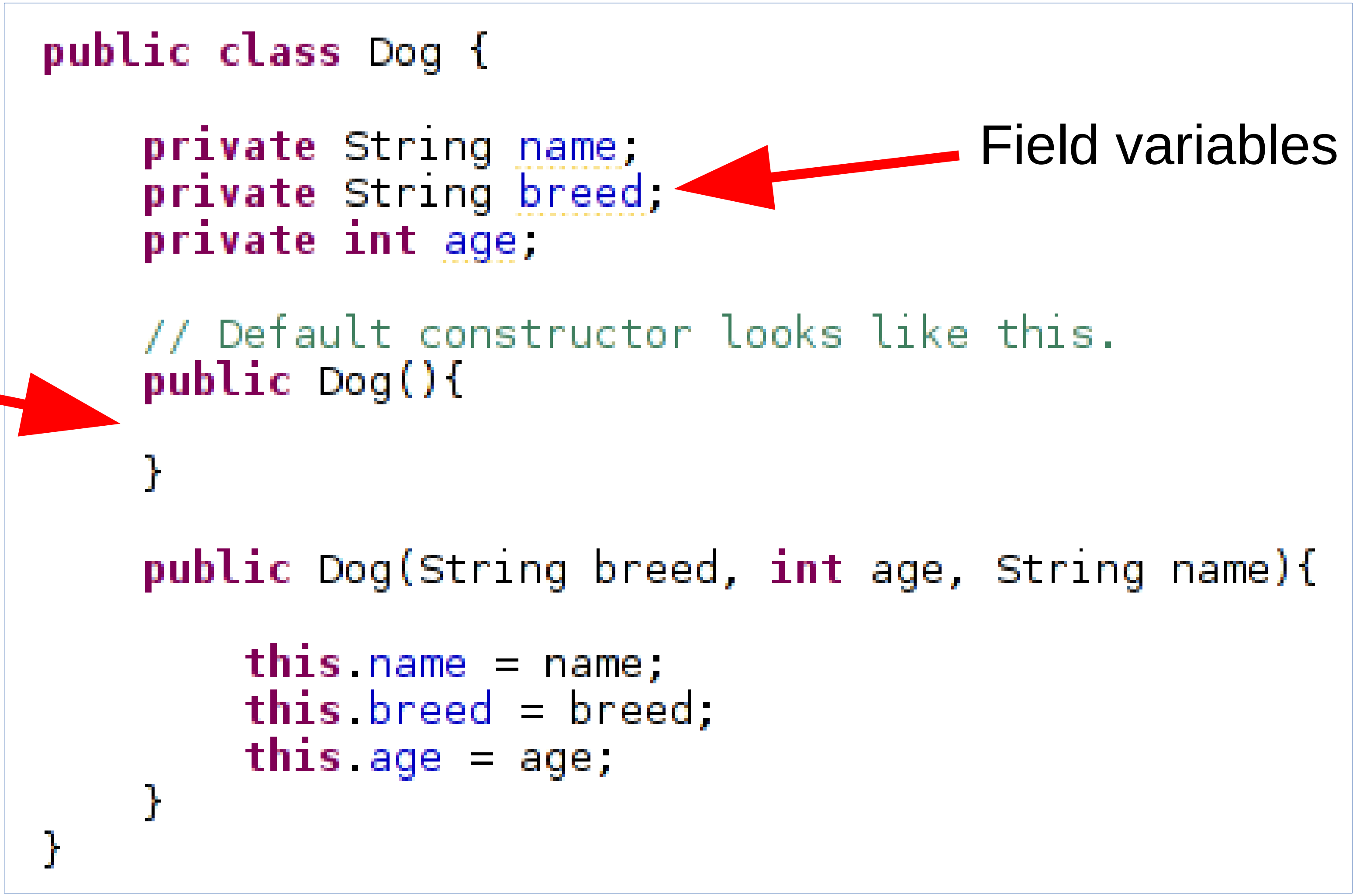
Name: Snow
Breed: Husky
Age: 6

What is a constructor?

- As the name suggests, a constructor *constructs* an instance of a class.
- It is a method *without* a return type, *the same name* as the class and possibly attributes (i.e. public Dog(*String breed, int age, String Name*).
- It tells the *JVM* to allocate memory for an instance of the class and initialize it.
- Every class by default has a *default* constructor that takes no arguments.

Example

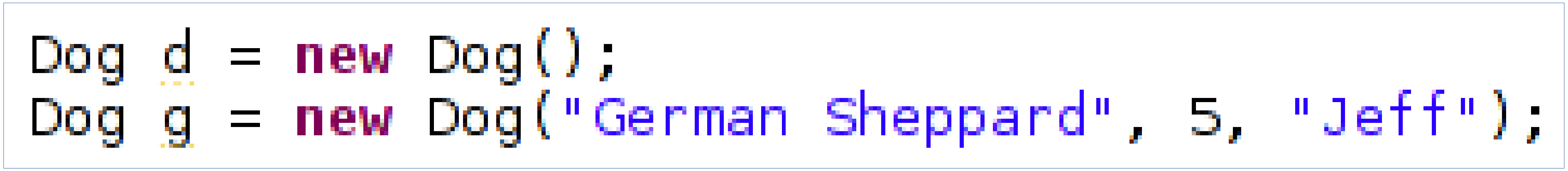
This *default* constructor is not actually shown by default, but is implied as shown.



```
public class Dog {

    private String name;
    private String breed;
    private int age;

    // Default constructor looks like this.
    public Dog(){

    }

    public Dog(String breed, int age, String name){

        this.name = name;
        this.breed = breed;
        this.age = age;
    }
}
```

Field variables

Building an object



```
Dog d = new Dog();
Dog g = new Dog("German Sheppard", 5, "Jeff");
```

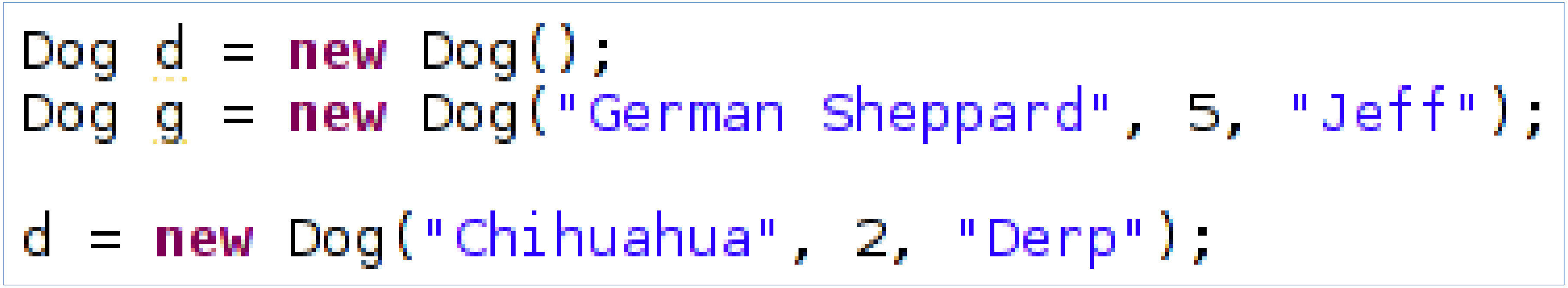
- *Dog d* and *Dog g* mean we want to create *reference* variables of type *Dog* called *d* and *g* referring to instances of *Dog.*

- new Dog() is using our first constructor, and the other our second constructor.

- If you implement a constructor with arguments, the implied default constructor disappears. It is only still here as it is *explicitly defined* in the code.

Building an object

- What happens if we set our *Dog* reference variable *d* to something else?



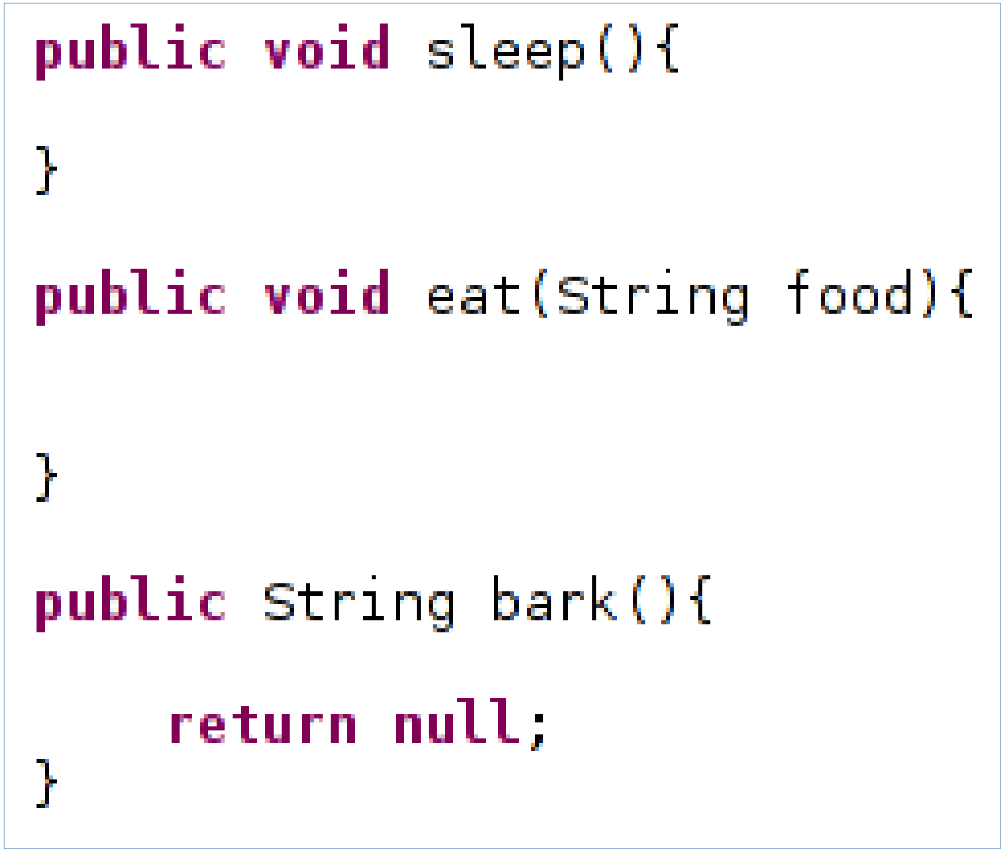
```
Dog d = new Dog();
Dog g = new Dog("German Sheppard", 5, "Jeff");

d = new Dog("Chihuahua", 2, "Derp");
```

- *d* was the only thing referring to our first instance of *Dog*.

- Therefore, since *d* now refers to a new instance of *Dog*, the old instance will be *garbage collected* and the memory it is using will be *freed*.

Instance Methods

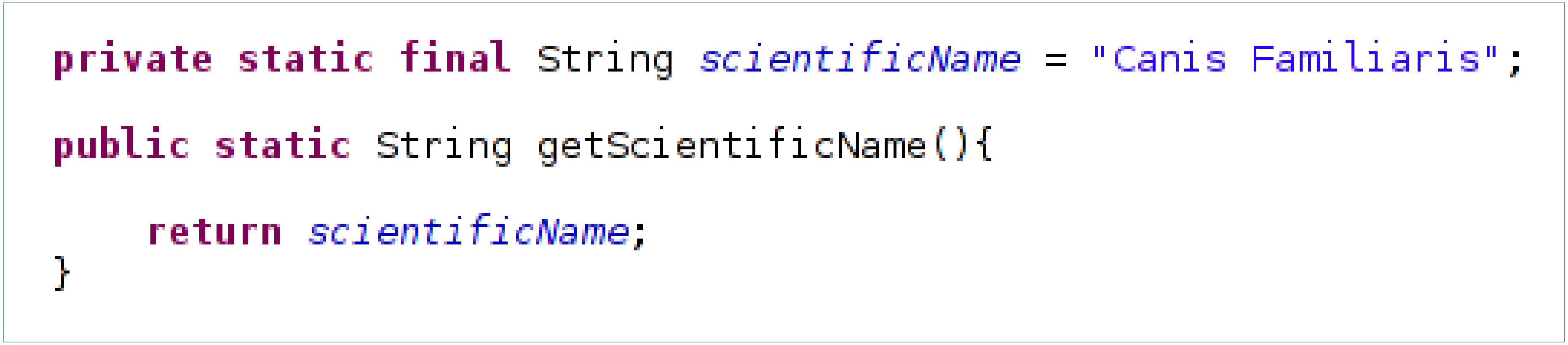
- These methods are only accessible from *instances* of the object.



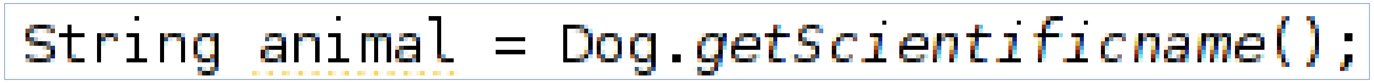
```
public void sleep(){

}

public void eat(String food){


}

public String bark(){

    return null;
}
```

Static Methods

- These methods are accessible from the class itself.
- They are denoted by the *static* keyword in the method declaration.
- Any methods or variables used within static methods must also be static.



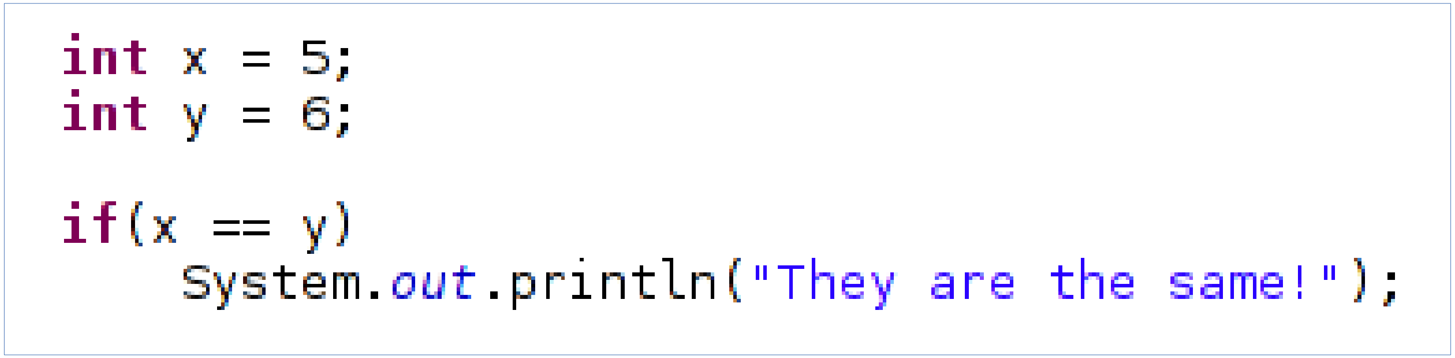
```
private static final String scientificName = "Canis Familiaris";

public static String getScientificName(){

    return scientificName;
}
```



```
String animal = Dog.getScientificname();
```

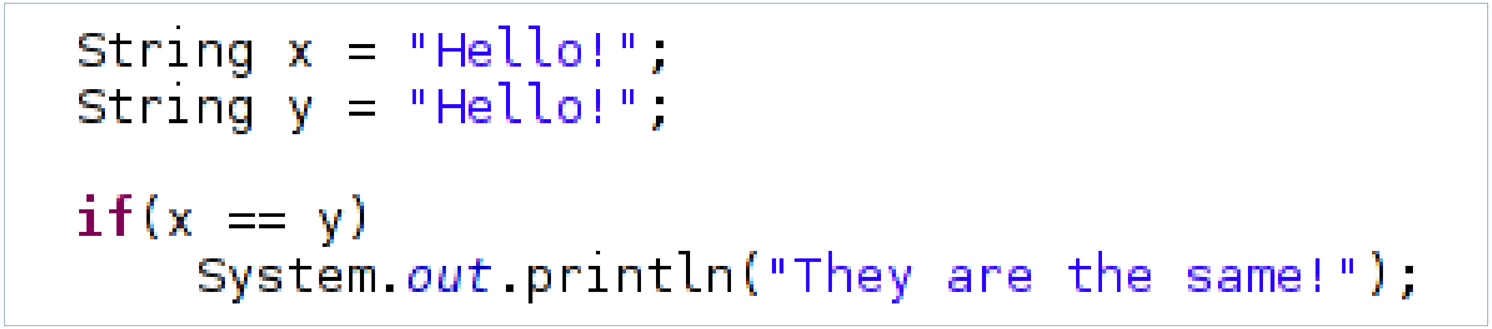
Checking Equality

- We can check the equality of primitives using the == boolean operator.



```
int x = 5;
int y = 6;

if(x == y)
    System.out.println("They are the same!");
```

- However, with strings, the following would not print anything.



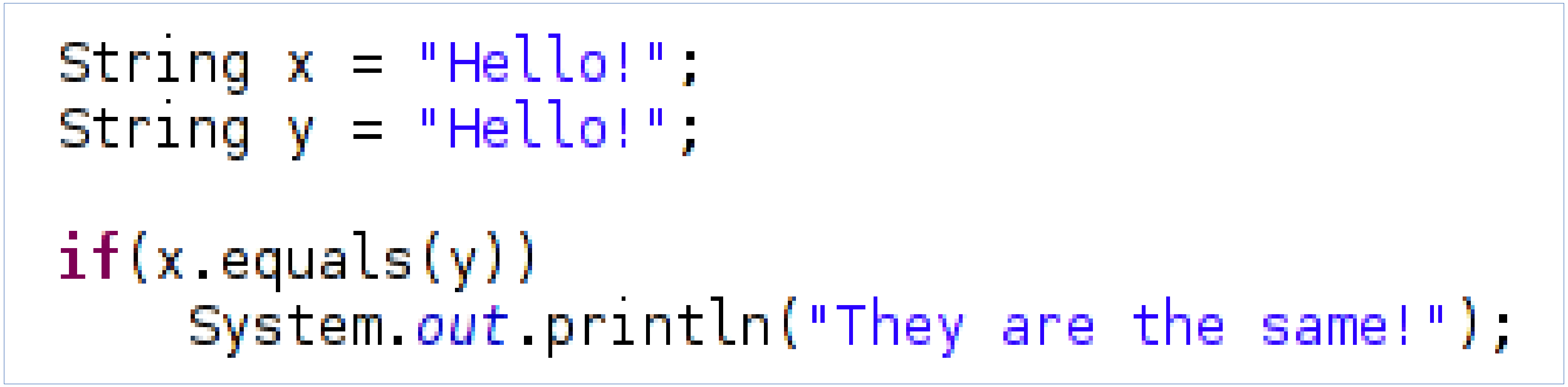
```
String x = "Hello!";
String y = "Hello!";

if(x == y)
    System.out.println("They are the same!");
```

What is wrong?

- == *only* works as expected for *primitive* data types.
- When used with *reference* variables, all it does is check if they point to the same memory location.
- String is a class, meaning any instances (i.e. x and y) will refer to the memory location where those strings start.

Object Equality

- *Data* equality of *reference* variables is checked with the .equals method inherited from Java's default top level class *Object*.



```
String x = "Hello!";
String y = "Hello!";

if(x.equals(y))
    System.out.println("They are the same!");
```

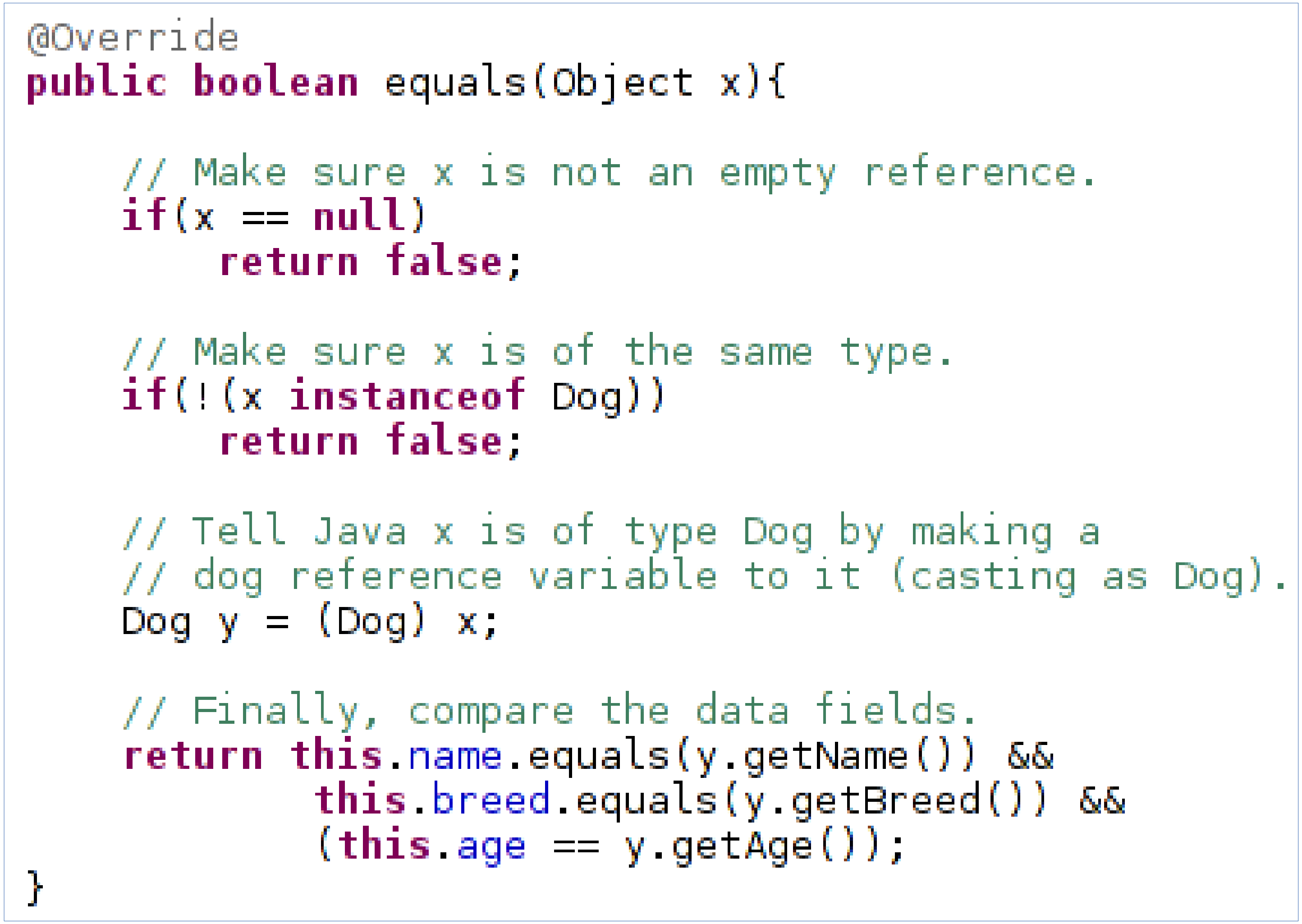
Defining Equals

- .equals is already defined for many built in Java classes like String, Integer, BigInteger. etc

- For our own classes, we must *override* it, or it will automatically default back to *Object*'s equals, which works just like ==.

Defining Equals

@Override means we are overriding the default *equals* method provided by *Object*



```
@Override
public boolean equals(Object x){

    // Make sure x is not an empty reference.
    if(x == null)
        return false;

    // Make sure x is of the same type.
    if(!(x instanceof Dog))
        return false;

    // Tell Java x is of type Dog by making a
    // dog reference variable to it (casting as Dog).
    Dog y = (Dog) x;

    // Finally, compare the data fields.
    return this.name.equals(y.getName()) &&
            this.breed.equals(y.getBreed()) &&
            (this.age == y.getAge());
}
```

Things to remember

- Always override equals, rather than making your own equality checking method.
- Other standard Java datatypes *expect* equals to exist and be overridden to work properly with your datatypes (i.e. List, Map, Set. etc)
- When overriding .equals, make sure to always check within the method and make sure the argument is:
  - Of the same type (use ***instanceof***)
  - Not null

Overriding hashCode

- Whenever you override equals, it is *good practice* to override the .hashCode method inherited from *Object* as well.

- Certain useful and common standard Java classes use *hashing* to work efficiently with your data structures.

What is hashing?

- *Hashing* in Java is the process of generating a short irreversible code based on the content of a data structure.

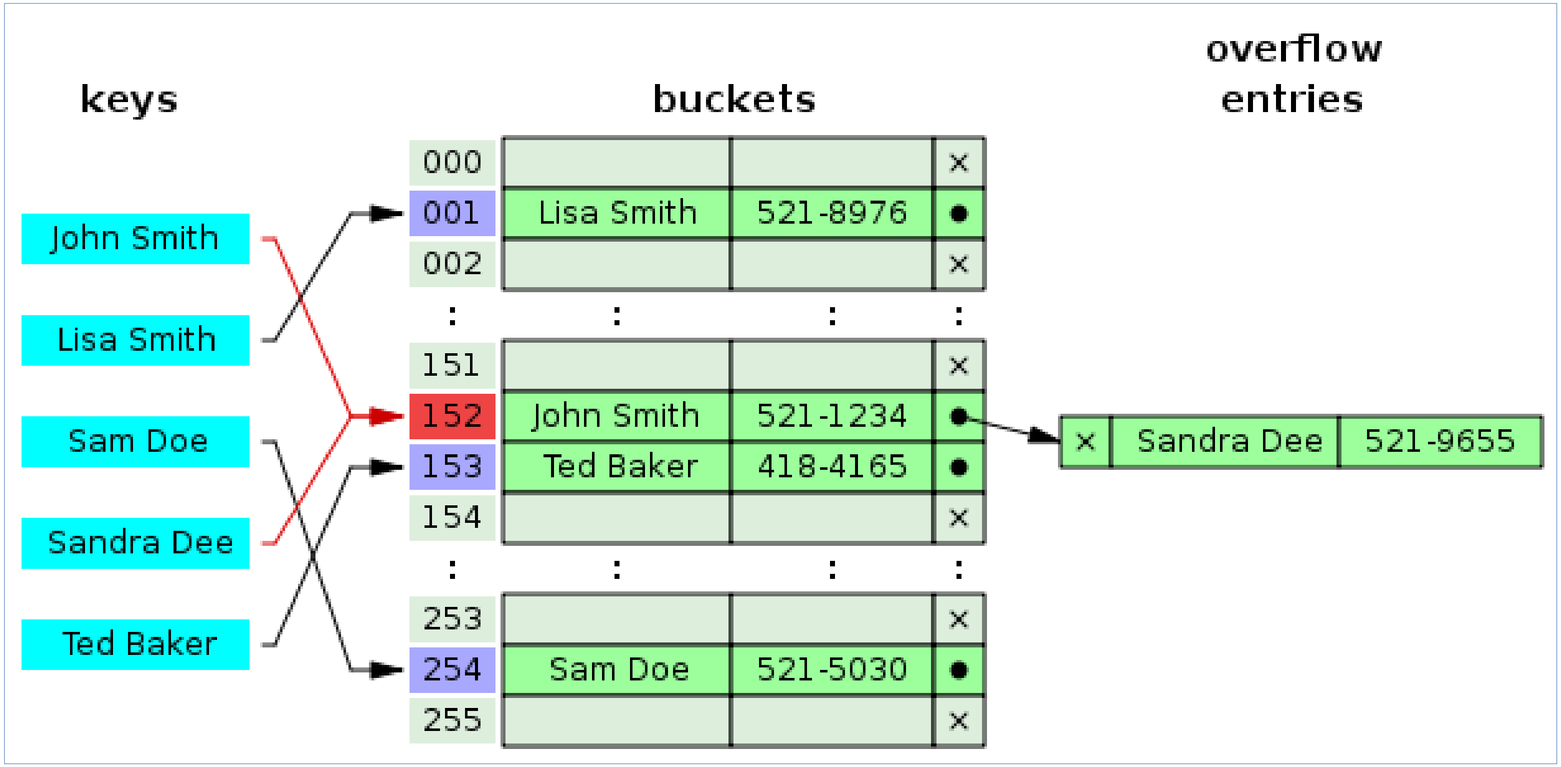
- When the hashes produced by a data structure's hashing function are *mostly* unique, hashing data structures are more efficient.

Example: HashMap

- A *HashMap* is a collection of data, like an array, but uses *keys* to access data, rather than positional indices.

- The following maps names to phone numbers.

- Names with the same hash code cause a “collision” and form a chain.

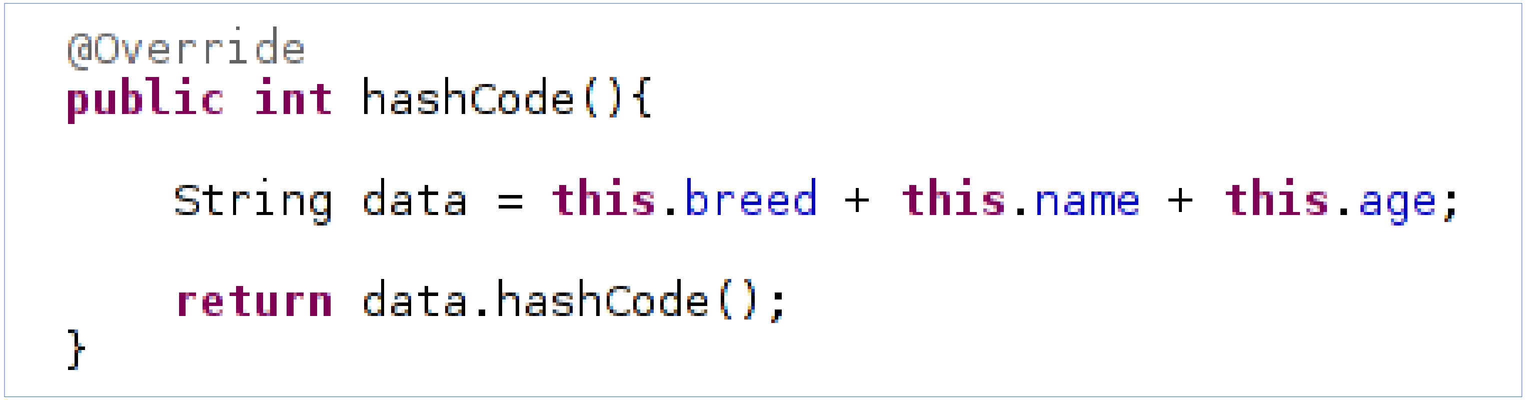


000			×
001	Lisa Smith	521-8976	●
002			×
:	:	:	:
151			×
152	John Smith	521-1234	●
153	Ted Baker	418-4165	●
154			×
:	:	:	:
253			×
254	Sam Doe	521-5030	●
255			×

Simple Hashing

- *Hashing* is a complicated field of computer science all on its own.

- For simple data structures, making a big string from the field data and using the built in hash is usually sufficient.



```
@Override
public int hashCode(){

    String data = this.breed + this.name + this.age;

    return data.hashCode();
}
```

Enumeration

- In our original *Dog class* example, we marked distinct breeds using a *String* property called *breed.*

- Using a string to represent breeds can lead to problems with data, like spelling mistakes and inconsistency (i.e. “germanshepherd”, “German Shepherd”. Etc).

- This makes equality checking difficult.

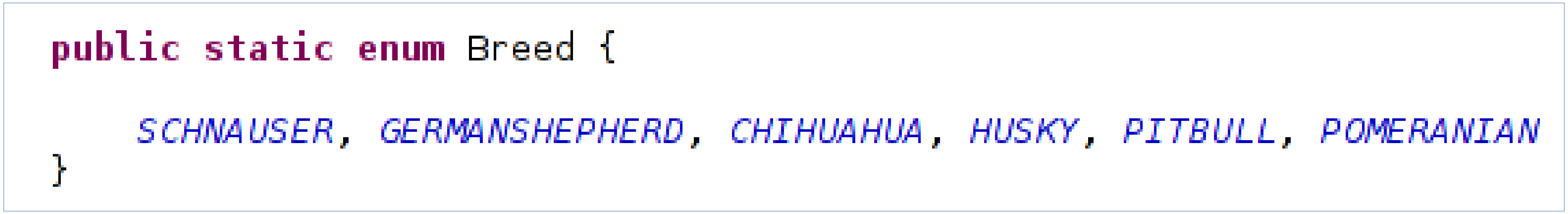
- There are a finite number of dog breeds out there, so we can fix this using an *enumeration*.

What is an enumeration?

- An *enumeration* is a simple datatype with a ***finite*** number of elements.

- Used for representing states, types. etc

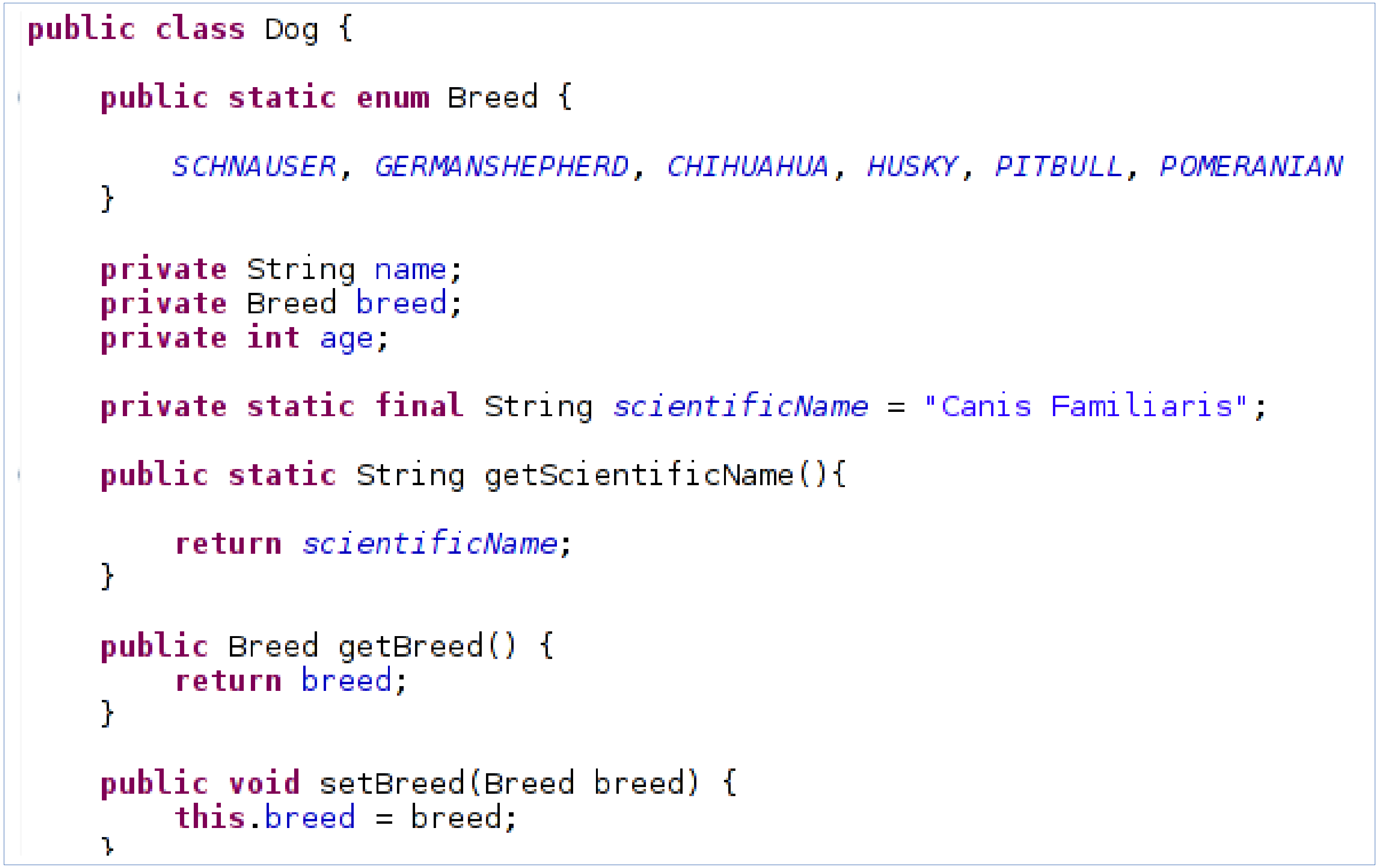
- Better than using a constant number, as the code does not change when more elements are added to the enumeration.



```
public static enum Breed {

    SCHNAUSER, GERMANSHEPHERD, CHIHUAHUA, HUSKY, PITBULL, POMERANIAN
}
```

Enumerated Class

- *Breed* stored as an enumerated class. Obviously, this is only a subset of the number of dog breeds out there.



```java
public class Dog {

    public static enum Breed {

        SCHNAUSER, GERMANSHEPHERD, CHIHUAHUA, HUSKY, PITBULL, POMERANIAN
    }

    private String name;
    private Breed breed;
    private int age;

    private static final String scientificName = "Canis Familiaris";

    public static String getScientificName(){

        return scientificName;
    }

    public Breed getBreed() {
        return breed;
    }

    public void setBreed(Breed breed) {
        this.breed = breed;
    }
```

Abstraction

- Sometimes, we want to express an *inheritance* relationship between classes.

- Back to our *Dog* example; perhaps we would like to define properties and methods for particular *breeds*.

- Obviously, we would still like properties and methods consistent with *all* dogs to apply to our breed, but we would also like to avoid redefining all that information.

Abstraction

- Java provides two mechanisms for doing this:
    - Abstract classes
    - Interfaces

Abstract Classes

- An abstract class is a class that cannot have instances of itself created.

- It defines a *partial* class that must be extended by some other class to be complete.

- Uses the *abstract* keyword to define itself and properties of itself that *must* be implemented by the extending class.

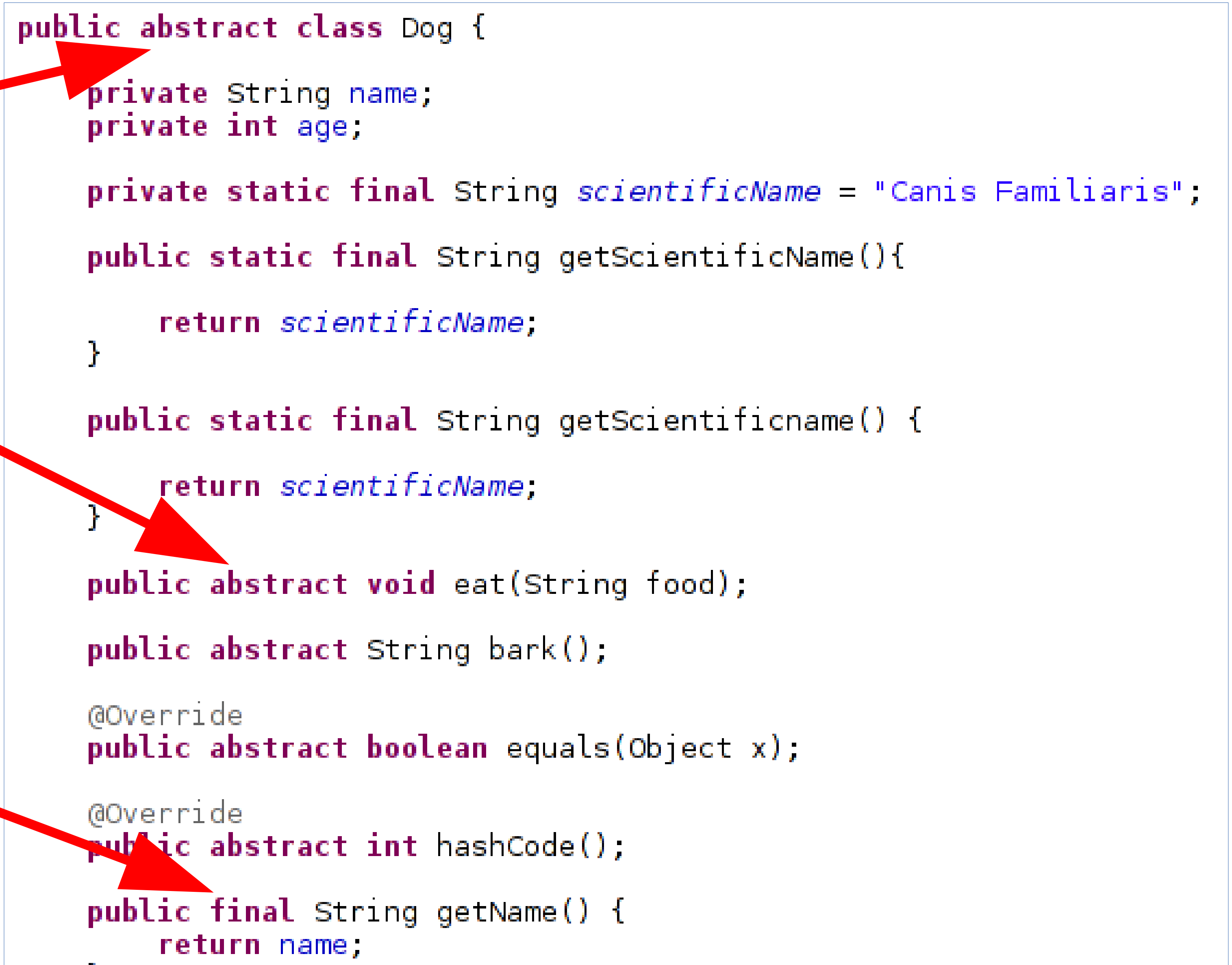
- Lets turn *Dog* into an abstract class.

Abstract Classes

Abstract class declaration

abstract **means classes extending this one** ***must*** **implement these methods.**

final **means these methods** ***cannot*** **be overridden.**



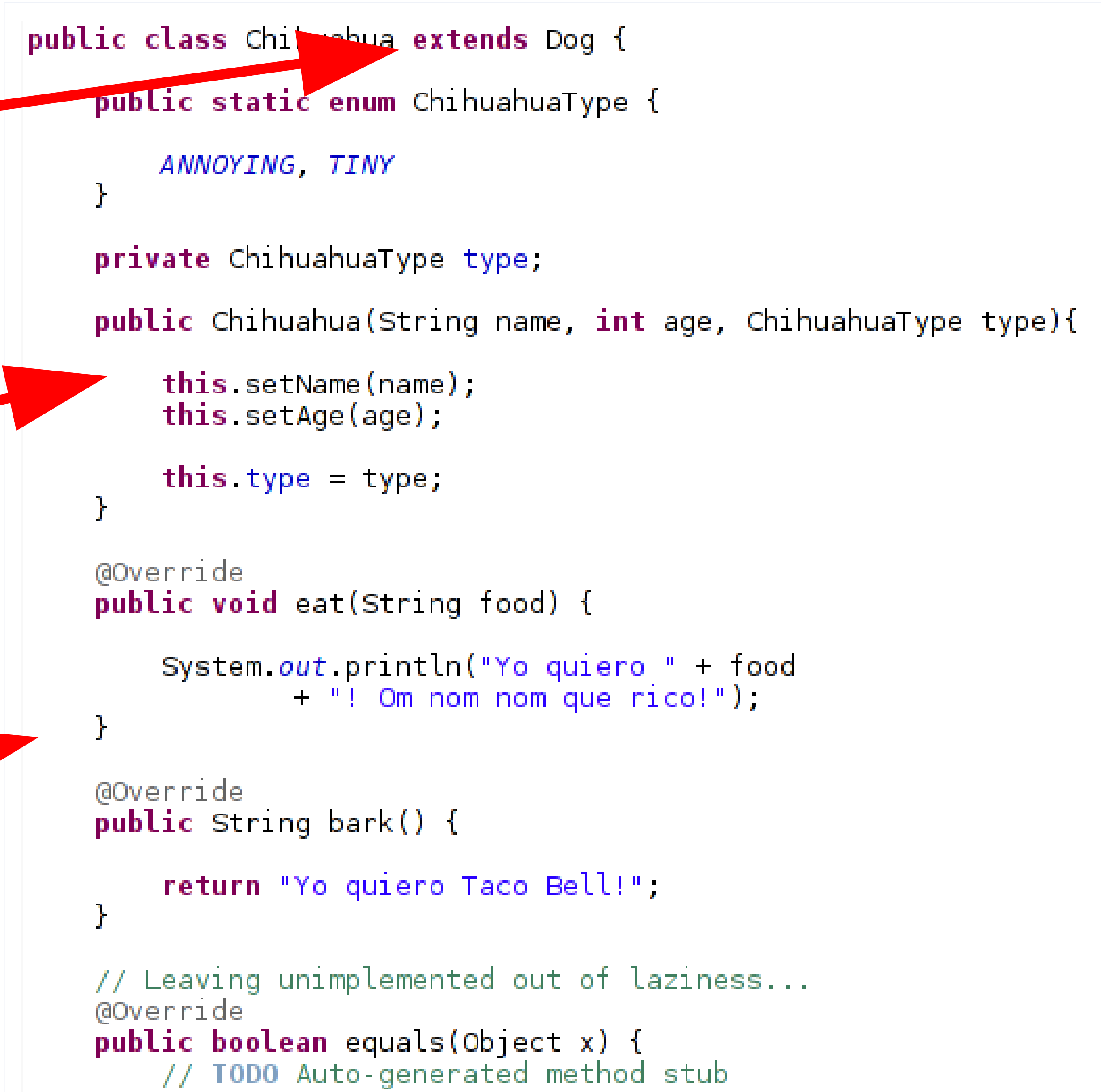
```
public abstract class Dog {

    private String name;
    private int age;

    private static final String scientificName = "Canis Familiaris";

    public static final String getScientificName(){

        return scientificName;
    }

    public static final String getScientificname() {

        return scientificName;
    }

    public abstract void eat(String food);

    public abstract String bark();

    @Override
    public abstract boolean equals(Object x);

    @Override
    public abstract int hashCode();

    public final String getName() {
        return name;
```

Abstract Classes

Abstract class *extends* declaration

Constructors can only be defined in extending classes, not in abstract ones

Overridden and implemented methods.



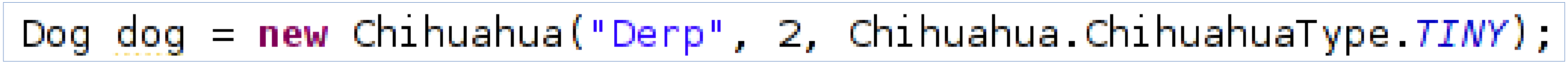
```
public class Chihuahua extends Dog {

    public static enum ChihuahuaType {

        ANNOYING, TINY
    }

    private ChihuahuaType type;

    public Chihuahua(String name, int age, ChihuahuaType type){

        this.setName(name);
        this.setAge(age);

        this.type = type;
    }

    @Override
    public void eat(String food) {

        System.out.println("Yo quiero " + food
                + "! Om nom nom que rico!");
    }

    @Override
    public String bark() {

        return "Yo quiero Taco Bell!";
    }

    // Leaving unimplemented out of laziness...
    @Override
    public boolean equals(Object x) {
        // TODO Auto-generated method stub
```

Abstract Classes

- A class can extend at most one other class.

- Normal classes can be extended too, in fact, all *Java* classes are extended from the base *Object* class implicitly.

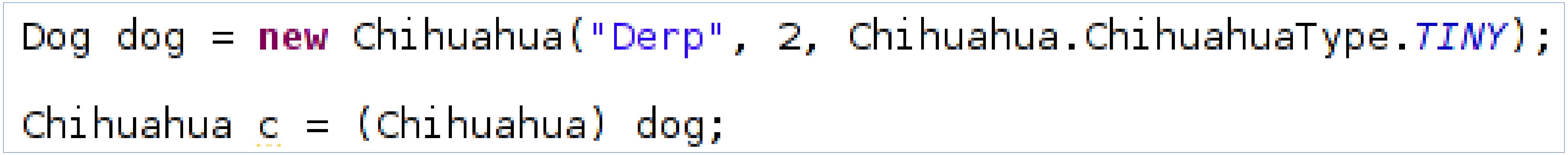
- Classes extending another are of their own type *and* the type of their base class.



```
Dog dog = new Chihuahua("Derp", 2, Chihuahua.ChihuahuaType.TINY);
```

Abstract Classes

- If the class *Chihuahua* defined methods <u>not</u> found in *Dog*, they would only be callable via *Chihuahua* type reference variables.



```
Dog dog = new Chihuahua("Derp", 2, Chihuahua.ChihuahuaType.TINY);

Chihuahua c = (Chihuahua) dog;
```

Interfaces

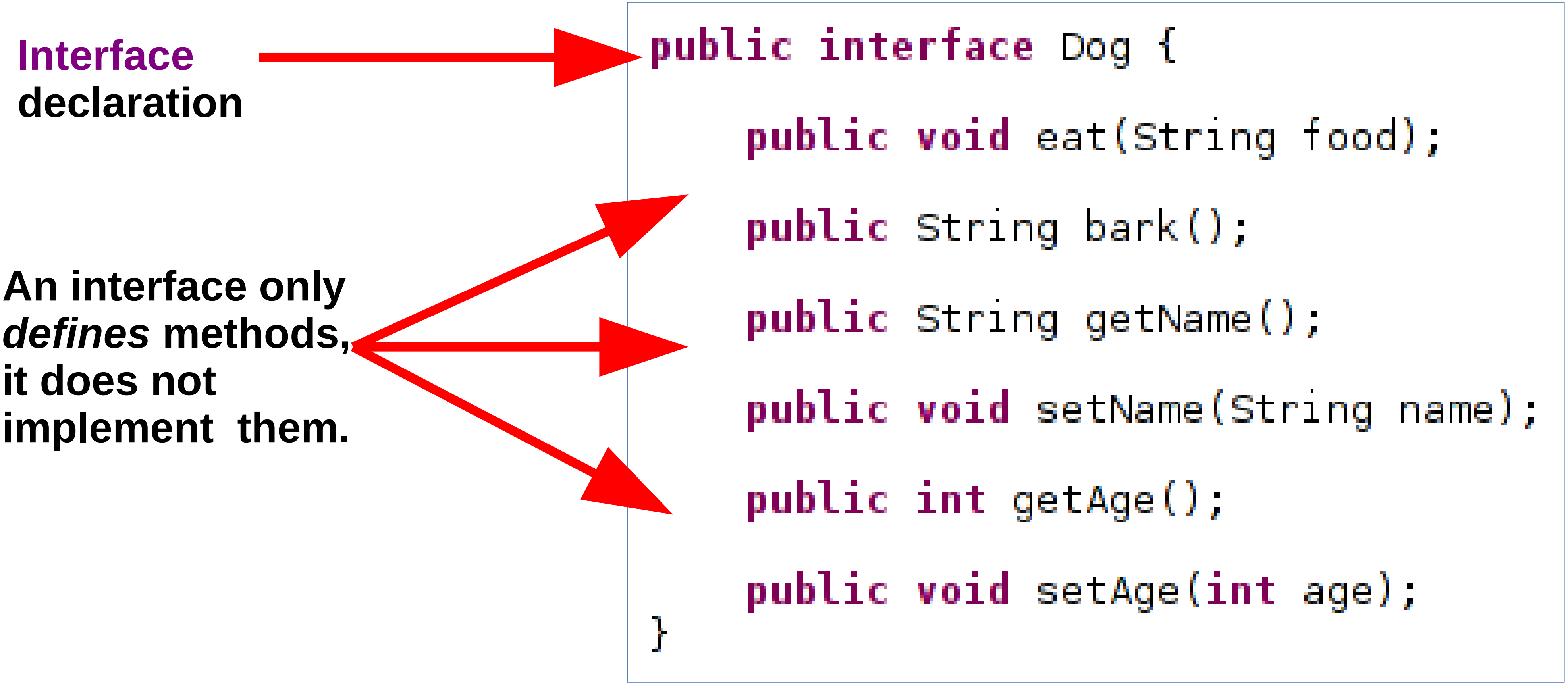
- An interface defines a *specification* or *contract* that a class must meet to be defined as an *instance* of that interface.

- Interfaces are not classes, so they cannot be instantiated.

- Everything in the interface must be implemented by the class using it.

- Classes can implement an unlimited number of interfaces.

Interfaces



Interface declaration

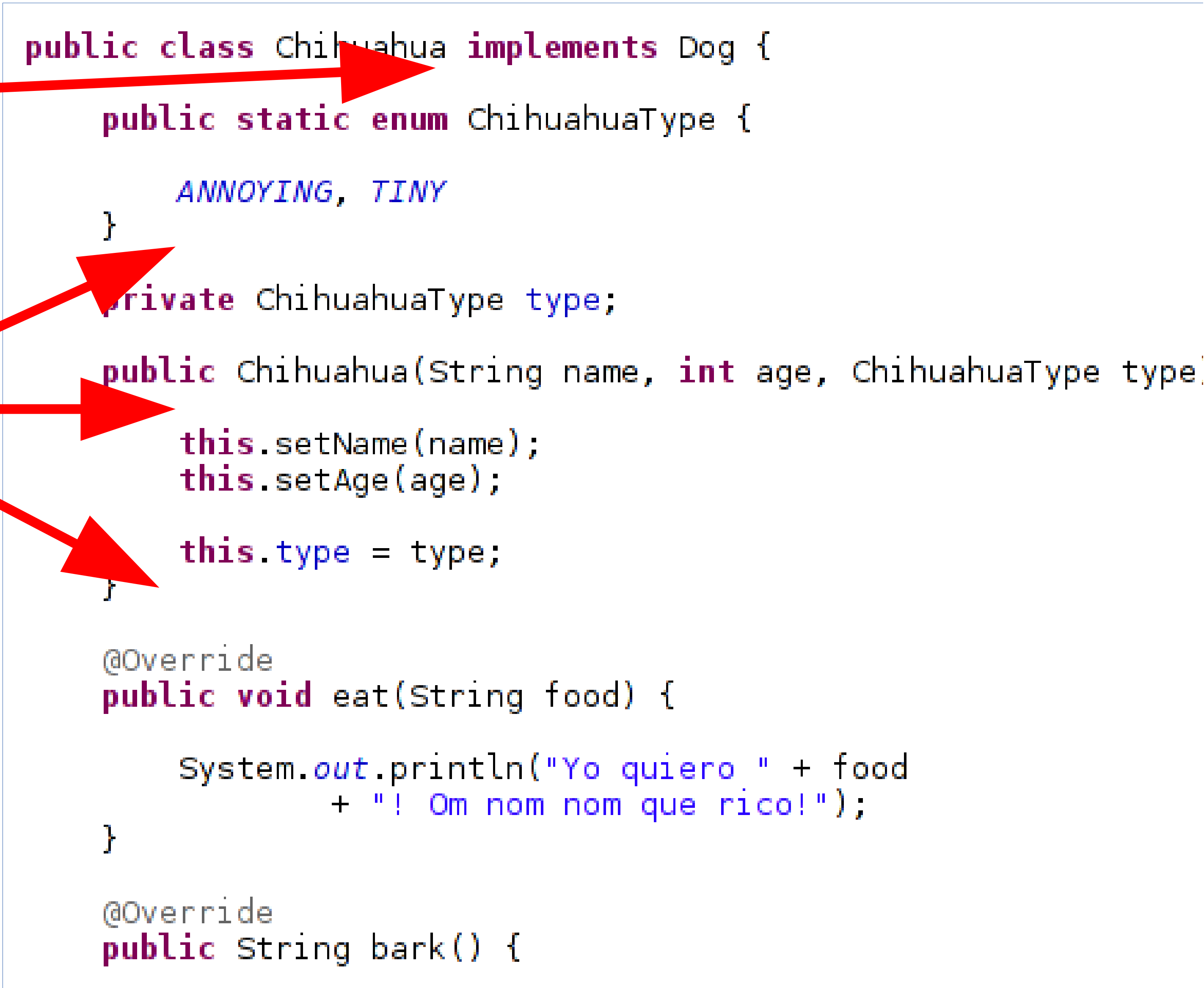
An interface only *defines* methods, it does not implement them.

```
public interface Dog {

    public void eat(String food);

    public String bark();

    public String getName();

    public void setName(String name);

    public int getAge();

    public void setAge(int age);
}
```

Interfaces

Interface implementation declaration

The rest is similar to what we saw for the abstract class example.



```java
public class Chihuahua implements Dog {

    public static enum ChihuahuaType {

        ANNOYING, TINY
    }

    private ChihuahuaType type;

    public Chihuahua(String name, int age, ChihuahuaType type)

        this.setName(name);
        this.setAge(age);

        this.type = type;
    }

    @Override
    public void eat(String food) {

        System.out.println("Yo quiero " + food
                + "! Om nom nom que rico!");
    }

    @Override
    public String bark() {
```

When to use one or the other?

- Use *abstract classes* or normal extension when you are defining a generalized object with methods and variables of its own.

- Use *interfaces* when defining a *capability* classes can have (i.e. *PrintWriter, BufferedWriter* and *FileWriter* all implement the *Writer* interface and supports methods like *write* and and *append)*

- In this example, our *Dog* class made more sense as an *abstract class.*

The End

The End :)