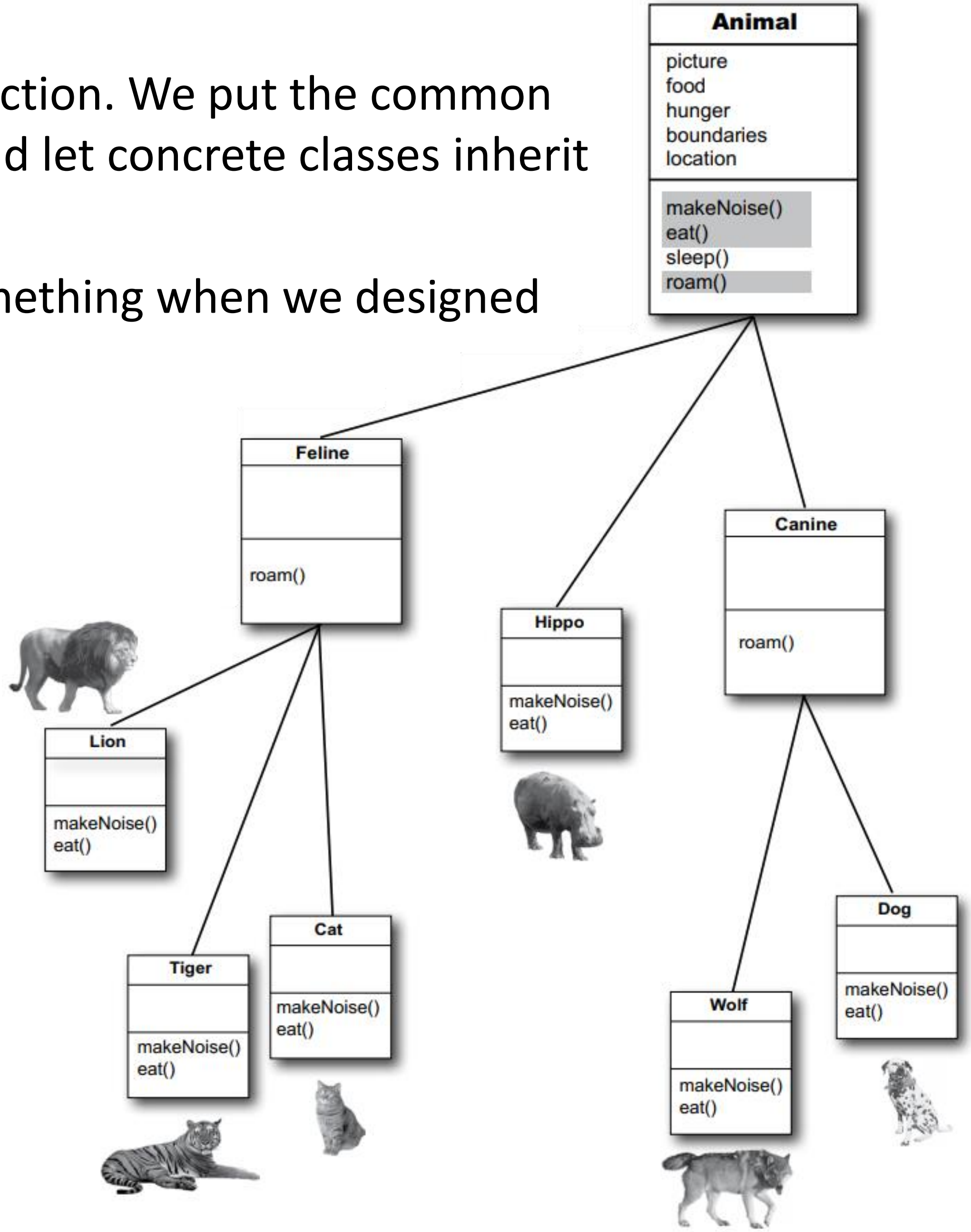
Chapter 8

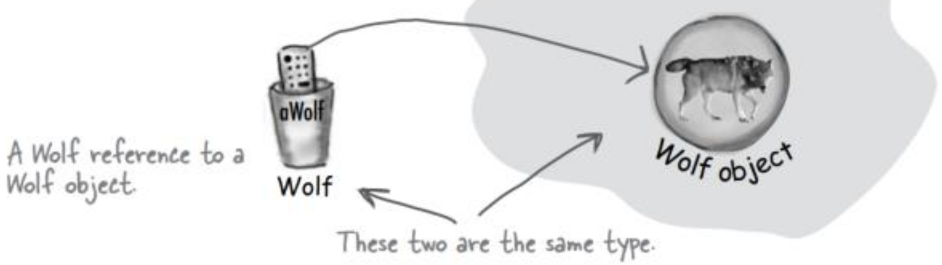
more about polymorphism

- Recall the idea of abstraction. We put the common things in a superclass and let concrete classes inherit it.
- Did we forget about something when we designed this?



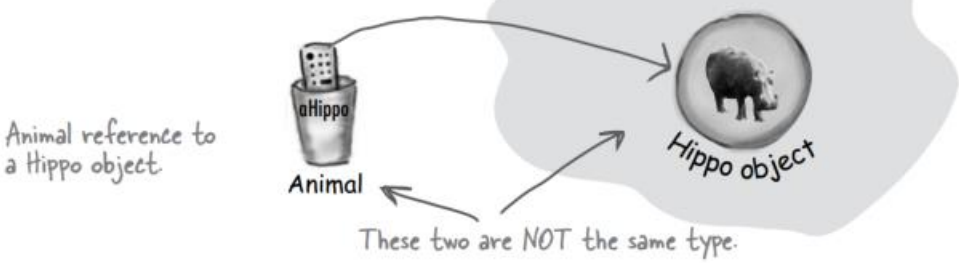
Hmm...make sense.

```
Wolf aWolf = new Wolf();
```


aWolf
Wolf
A Wolf reference to a Wolf object.
Wolf object
These two are the same type.

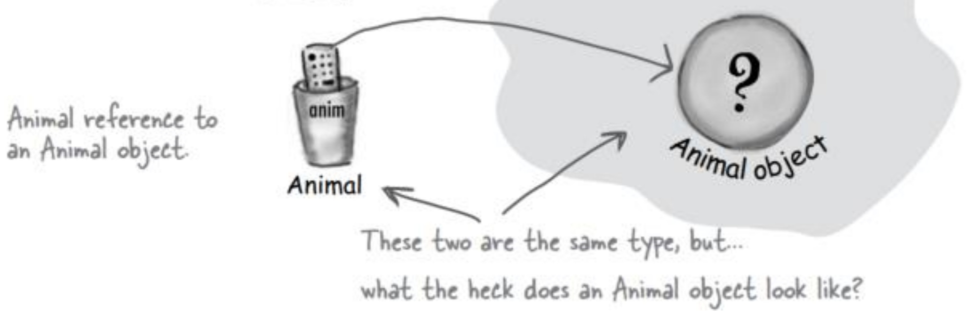
Hmm...we know we can do that.

```
Animal aHippo = new Hippo();
```

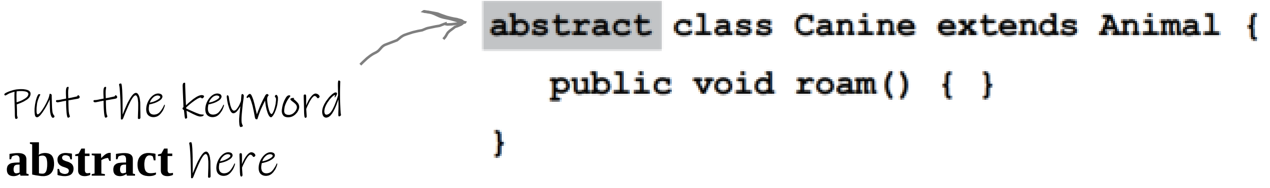

aHippo
Animal
Animal reference to a Hippo object.
Hippo object
These two are NOT the same type.

This is weird.

```
Animal anim = new Animal();
```


anim
Animal
Animal reference to an Animal object.
?
Animal object
These two are the same type, but...
what the heck does an Animal object look like?

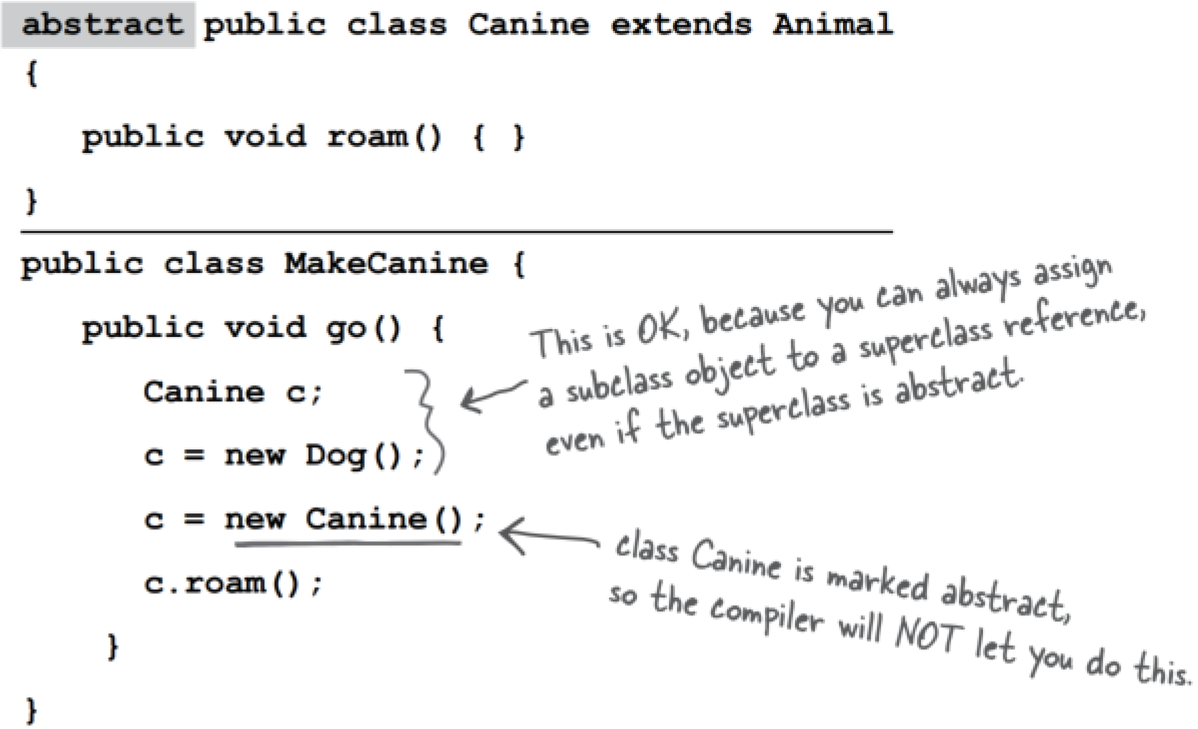
- What does a new Animal() object look like?
- What exactly is an Animal object? What shape is it? What color, size, number of legs...We don’t know.
- Some classes just should NOT be instantiated!
- We need an Animal class, for inheritance and polymorphism. But we don’t want a ‘real’ animal object.
- The compiler will stop any code, anywhere, from ever creating an instance of that type if we mark the class as abstract.



```
abstract class Canine extends Animal {
    public void roam() { }
}
```

Put the keyword **abstract** here

- An abstract class means that nobody can ever make a new instance of that class.
- You can still use that abstract class as a declared reference type, for the purpose of polymorphism



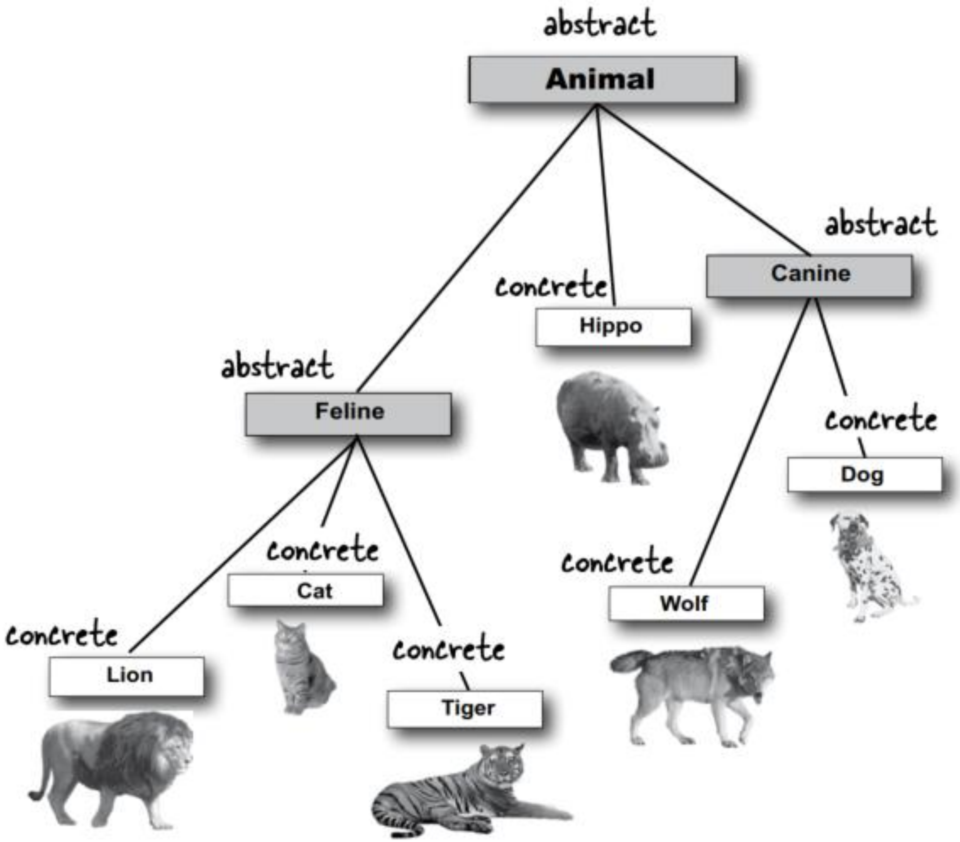
```
abstract public class Canine extends Animal
{
   public void roam() { }
}
```

```
public class MakeCanine {
   public void go() {
      Canine c;
      c = new Dog();
      c = new Canine();
      c.roam();
    }
}
```

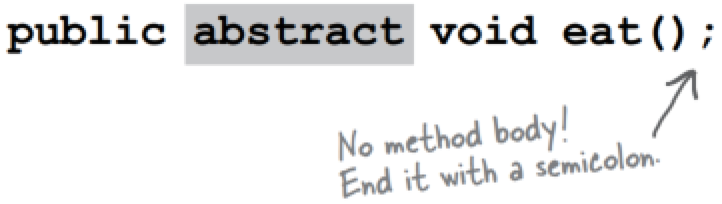
This is OK, because you can always assign a subclass object to a superclass reference, even if the superclass is abstract.

class Canine is marked abstract, so the compiler will NOT let you do this.

- When you’re designing your class inheritance structure, you have to decide which classes are abstract and which are concrete.
- Concrete classes are those that are specific enough to be instantiated. A concrete class just means that it’s OK to make objects of that type



- You can mark methods abstract, too. An abstract class means the class must be extended; an abstract method means the method MUST be overridden.
- An abstract method has no body. So no curly braces— just end the declaration with a semicolon.



```
public abstract void eat();
```

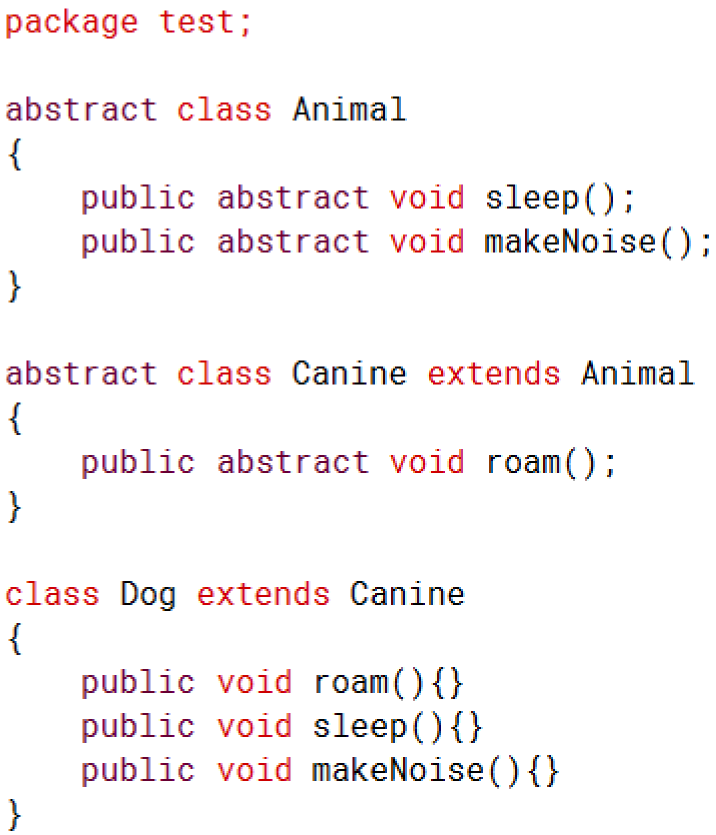
No method body!
End it with a semicolon.

- If you declare an abstract method, you MUST mark the class abstract as well. You can't have an abstract method in a non-abstract class. But you can mix both abstract and non-abstract methods in the abstract class.

- Abstract methods don’t have a body; they exist solely for polymorphism. That means the first concrete class in the inheritance tree must implement all abstract methods.
- If both Animal and Canine are abstract, for example, and both have abstract methods, class Canine does not have to implement the abstract methods from Animal.
- As soon as we get to the first concrete subclass, like Dog, that subclass must implement *all* of the abstract methods from both Animal and Canine.
- Remember that an abstract class can have both abstract and non-abstract methods, so Canine, for example, could implement an abstract method from Animal, so that Dog didn’t have to. But if Canine says nothing about the abstract methods from Animal, Dog has to implement all of Animal’s abstract methods.

OOP spirit:
The abstract class **defines** behavior, and the subclasses **implement** that behavior.

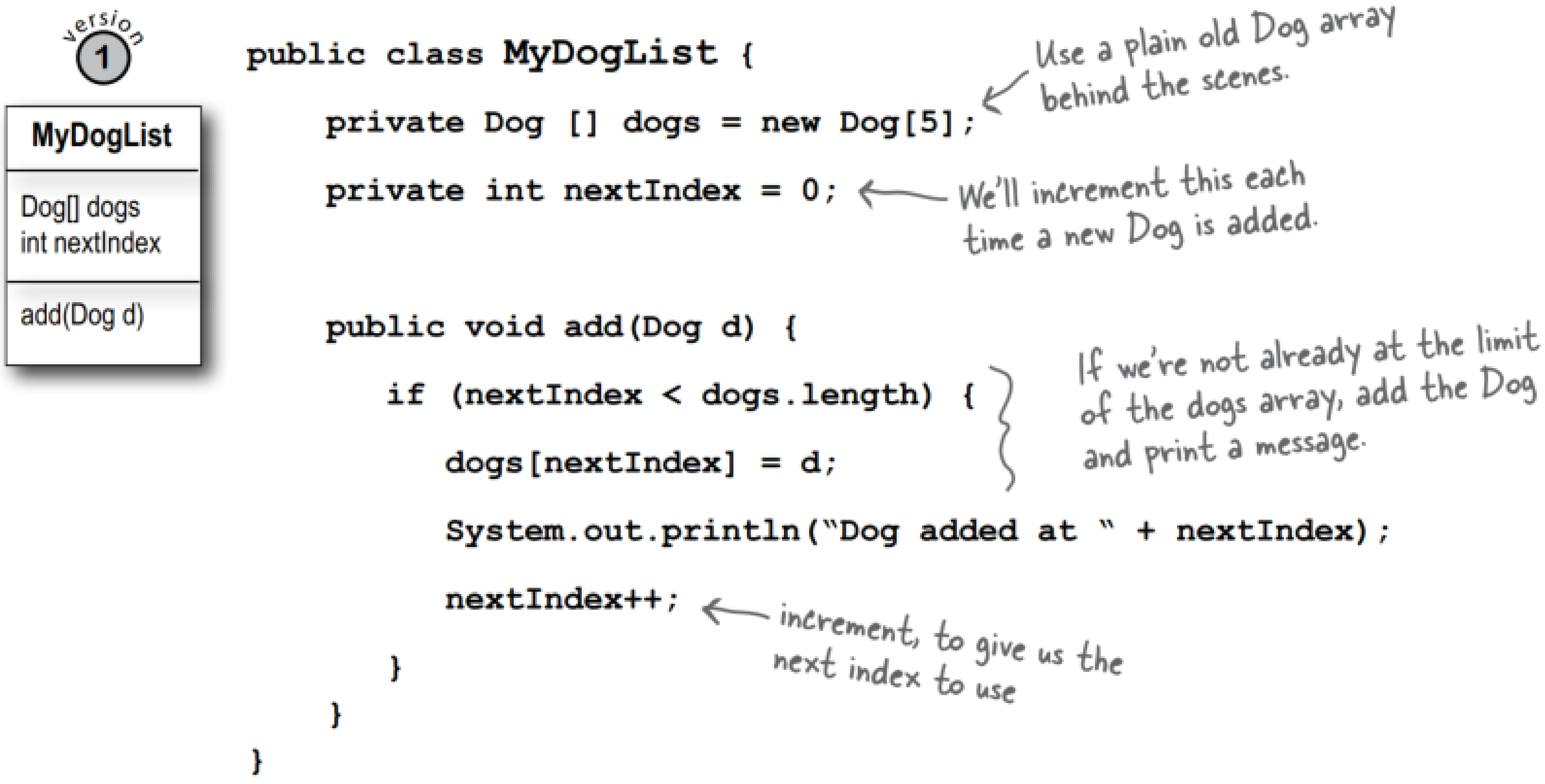
- Example



```
package test;

abstract class Animal
{
    public abstract void sleep();
    public abstract void makeNoise();
}

abstract class Canine extends Animal
{
    public abstract void roam();
}

class Dog extends Canine
{
    public void roam(){}
    public void sleep(){}
    public void makeNoise(){}
}
```

- Exercise: If you want to develop a series of “phone”, what’s your design?

- Example: write our own kind of list class, one that will hold Dog objects



version 1

MyDogList

Dog[] dogs
int nextIndex

add(Dog d)

```
public class MyDogList {

    private Dog [] dogs = new Dog[5];

    private int nextIndex = 0;


    public void add(Dog d) {

        if (nextIndex < dogs.length) {

            dogs[nextIndex] = d;

            System.out.println("Dog added at " + nextIndex);

            nextIndex++;

        }

    }

}
```

Use a plain old Dog array behind the scenes.

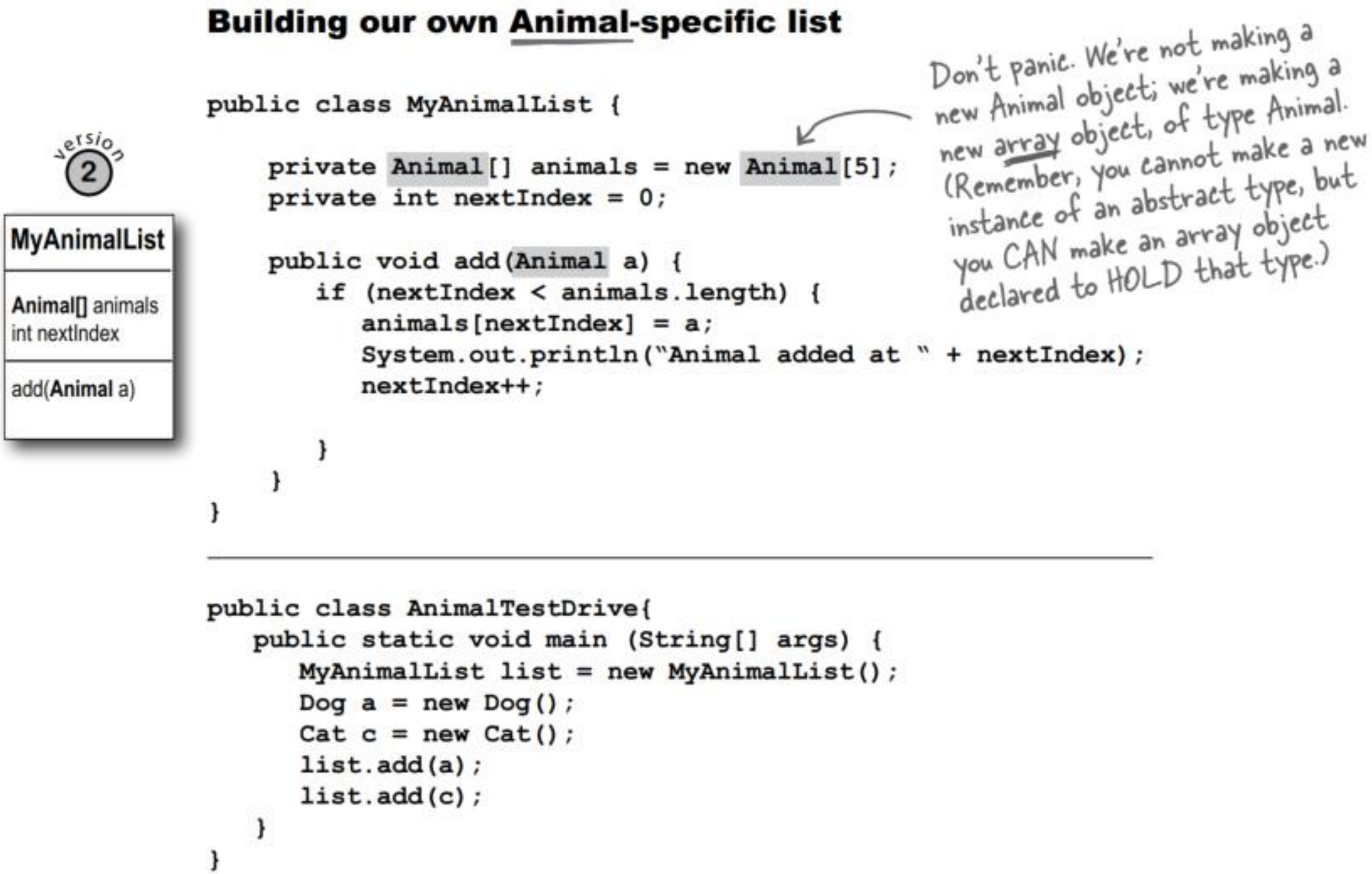
We'll increment this each time a new Dog is added.

If we're not already at the limit of the dogs array, add the Dog and print a message.

increment, to give us the next index to use

- What about we need a Cat list too?

1. Make a separate class, MyCatList, to hold Cat objects.
2. Make a single class, DogAndCatList, that keeps two different arrays as instance variables and has two different add() methods: addCat(Cat c) and addDog(Dog d).
3. Make heterogeneous AnimalList class, that takes any kind of Animal subclass
    - Design Animal class, the Dog class and Cat class are inherited from the Animal class.



Building our own Animal-specific list

MyAnimalList

Animal[] animals
int nextIndex

add(**Animal** a)

```
public class MyAnimalList {

    private Animal[] animals = new Animal[5];
    private int nextIndex = 0;

    public void add(Animal a) {
       if (nextIndex < animals.length) {
          animals[nextIndex] = a;
          System.out.println(“Animal added at ” + nextIndex);
          nextIndex++;

       }
    }
}
```

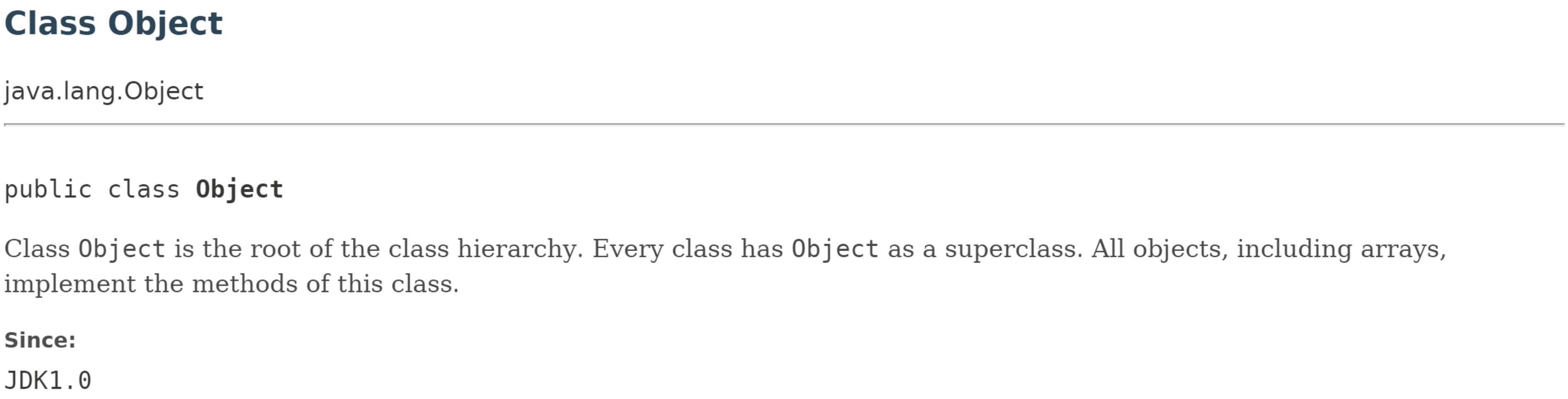
Don't panic. We're not making a new Animal object; we're making a new array object, of type Animal. (Remember, you cannot make a new instance of an abstract type, but you CAN make an array object declared to HOLD that type.)

```
public class AnimalTestDrive{
   public static void main (String[] args) {
      MyAnimalList list = new MyAnimalList();
      Dog a = new Dog();
      Cat c = new Cat();
      list.add(a);
      list.add(c);
   }
}
```

- Do we have a class which is generic enough to take anything?
- Take a look at the built-in class in Java: the **ArrayList** class. Look how the remove, contains, and indexOf method all use an object of type...***Object***!

ArrayList
boolean add(Object elem) Adds the object parameter to the list. **boolean remove(int index)** Removes the object at the index parameter. **boolean remove(Object elem)** Removes this object (if it’s in the ArrayList). **boolean contains(Object elem)** Returns ‘true’ if there’s a match for the object parameter **boolean isEmpty()** Returns ‘true’ if the list has no elements **int indexOf(Object elem)** Returns either the index of the object parameter, or -1 **int size()** Returns the number of elements currently in the list **Object get(int index)** Returns the object currently at the index parameter

- Every class in Java extends class Object. Class Object is the mother of all classes; it’s the superclass of everything
- Every class you write extends Object, without your ever having to say it
- Any class that doesn’t explicitly extend another class, implicitly extends Object.



Class Object

java.lang.Object

```
public class Object
```

Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class.

Since:

JDK1.0

- What’s in this ‘ultra-super-class’ Object?

① equals(Object o)

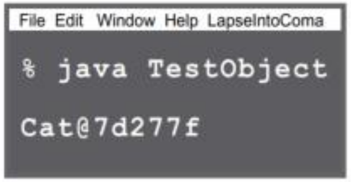
```
Dog a = new Dog();
Cat c = new Cat();

if (a.equals(c)) {
    System.out.println(“true”);
} else {
    System.out.println(“false”);
}
```

② getClass()

```
Cat c = new Cat();
System.out.println(c.getClass());
```

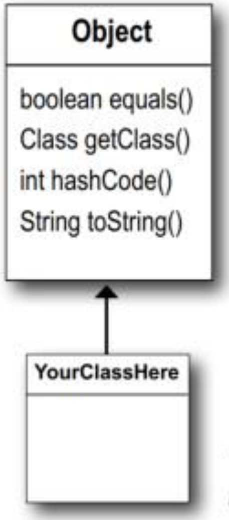
④ toString()

```
Cat c = new Cat();
System.out.println(c.toString());
```



```
% java TestObject
Cat@7d277f
```

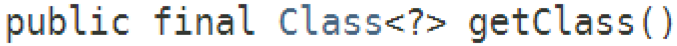
Prints out a String message with the name of the class and some other number we rarely care about.



Just SOME of the methods of class Object.

Every class you write inherits all the methods of class Object. The classes you’ve written inherited methods you didn’t even know you had.

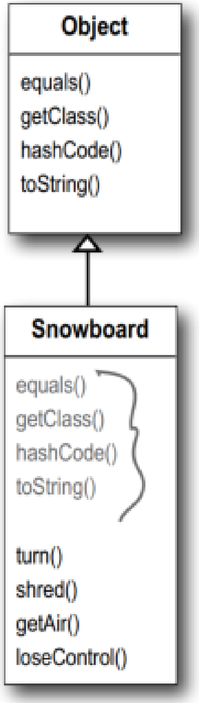
- Is class Object abstract?
- No, Object is a non-abstract class. It’s got method implementation code that all classes can inherit and use out-of-the-box, without having to override the methods.
- Can you override the methods in Object?
- Some of them are marked final, which means you can’t override them. You’re encouraged (strongly) to override hashCode(), equals(), and toString() in your own classes.



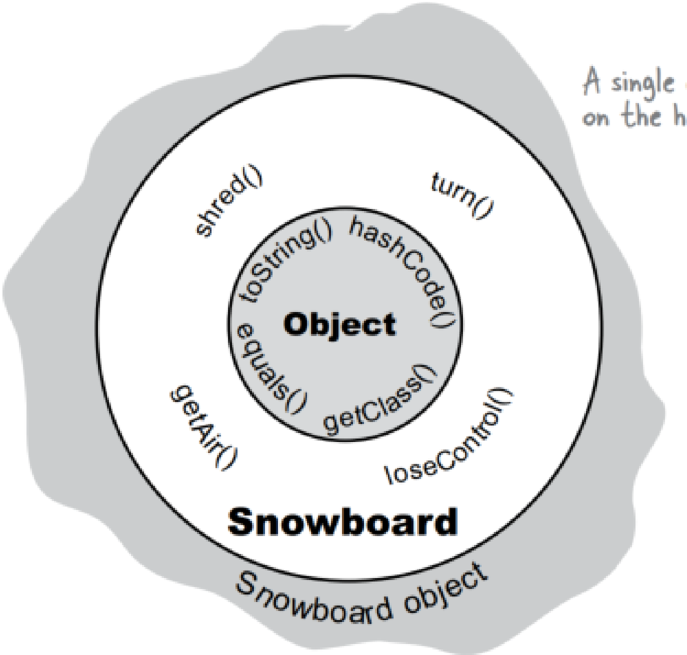
```
public final Class<?> getClass()
```

- Why Java allows to instantiate an Object object?
- The most common use of an instance of type Object is for thread synchronization (you’ll learn thread in chapter 15)
- Note: when objects are referred to by an Object reference type, Java thinks it’s referring to an instance of type Object. And that means the only methods you’re allowed to call on that object are the ones declared in class Object

- The compiler decides whether you can call a method based on the reference type, not the actual object type.
- Even if you know the object is a Dog object, the compiler sees it only as a generic Object, so you cannot invoke the methods that the Dog class has.

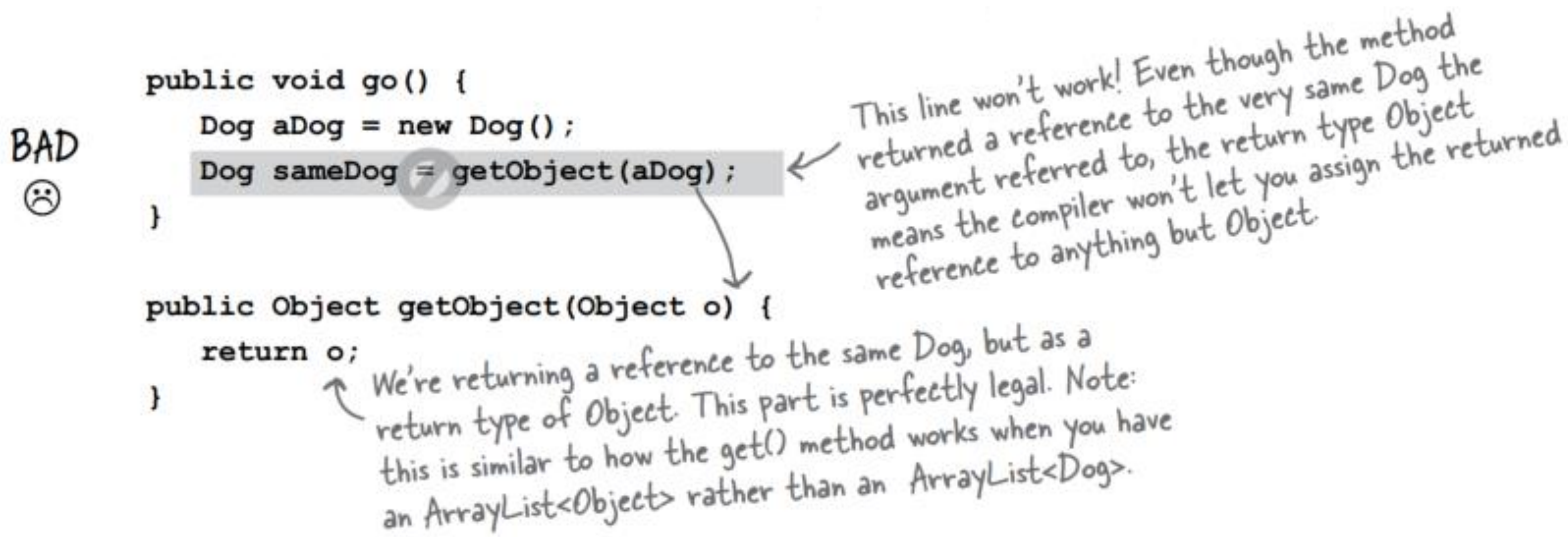


Snowboard inherits methods from superclass Object, and adds four more.



A single object on the heap.

There is only ONE object on the heap here. A Snowboard object. But it contains both the Snowboard class parts of itself and the Object class parts of itself.

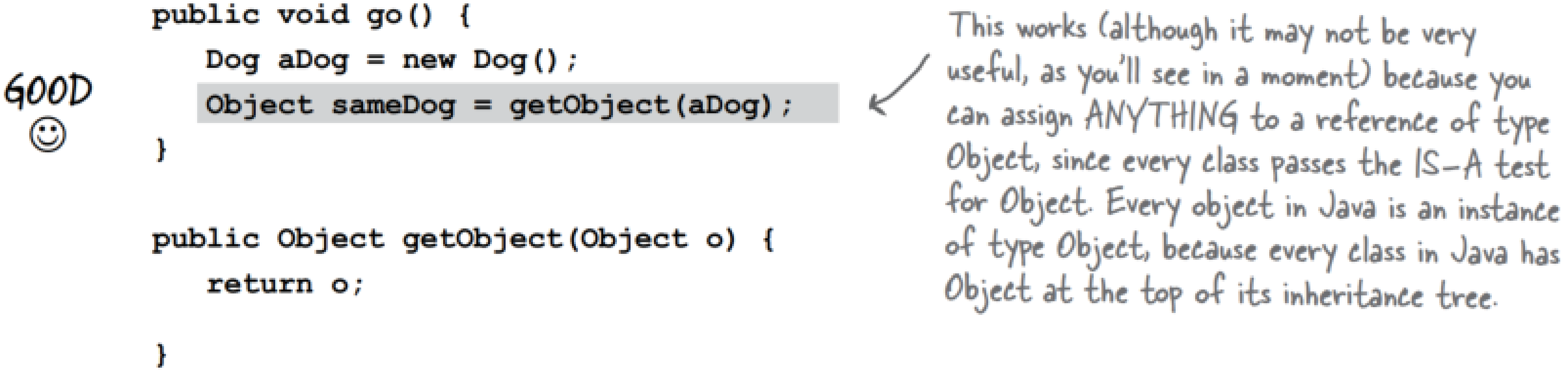


BAD ☹

```
public void go() {
    Dog aDog = new Dog();
    Dog sameDog = getObject(aDog);
}

public Object getObject(Object o) {
    return o;
}
```

This line won't work! Even though the method returned a reference to the very same Dog the argument referred to, the return type Object means the compiler won't let you assign the returned reference to anything but Object.

We're returning a reference to the same Dog, but as a return type of Object. This part is perfectly legal. Note: this is similar to how the get() method works when you have an ArrayList<Object> rather than an ArrayList<Dog>.



GOOD ☺

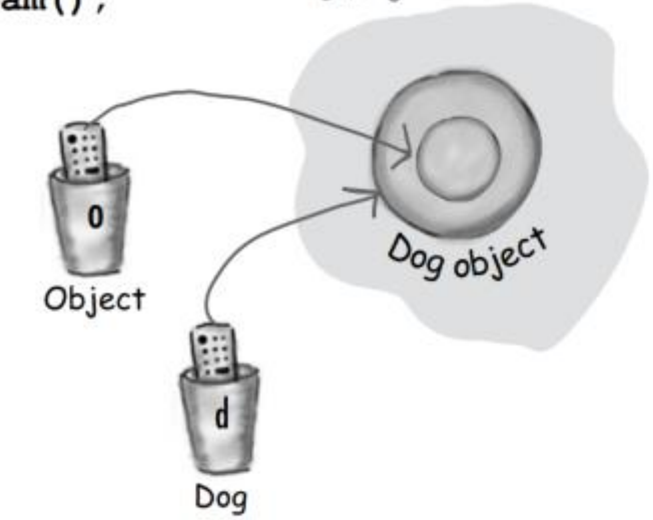
```
public void go() {
    Dog aDog = new Dog();
    Object sameDog = getObject(aDog);
}

public Object getObject(Object o) {
    return o;

}
```

This works (although it may not be very useful, as you'll see in a moment) because you can assign ANYTHING to a reference of type Object, since every class passes the IS-A test for Object. Every object in Java is an instance of type Object, because every class in Java has Object at the top of its inheritance tree.

- In fact, we know it is a Dog object but we can't invoke Dog-specific class methods because the compiler thinks it is the Object object.
- If you're sure* the object is really a Dog, you can make a new Dog reference to it by copying the Object reference, and forcing that copy to go into a Dog reference variable, using a cast (Dog)

```
Object o = al.get(index);
Dog d = (Dog) o;
d.roam();
```

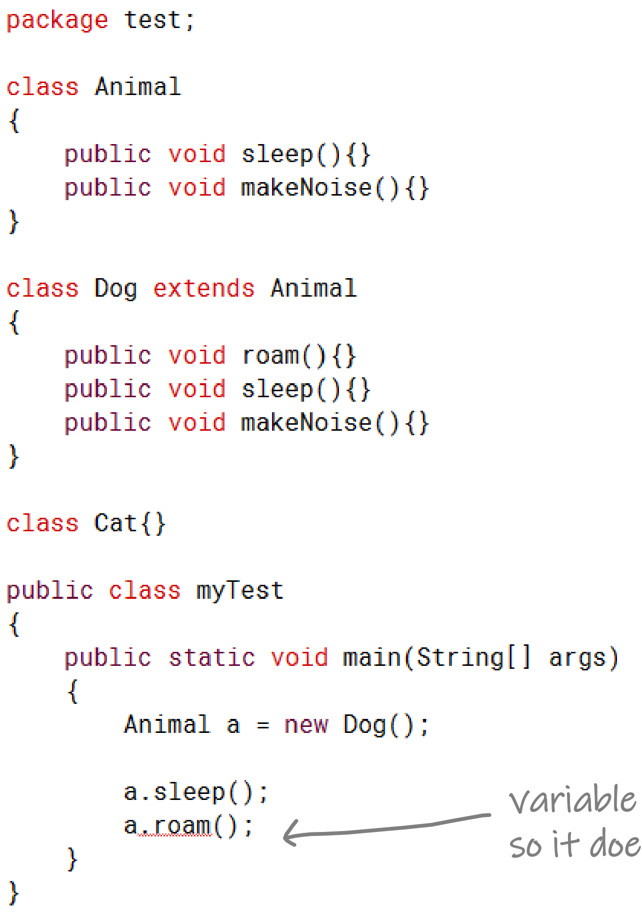
← cast the Object back to a Dog we know is there.



*If you're not sure it's a Dog, you can use the instanceof operator to check. Because if you're wrong when you do the cast, the program will crash in some way.

```
if (o instanceof Dog) {
   Dog d = (Dog) o;
}
```

- Just remember that the compiler checks the class of the ***reference variable***, not the class of the actual object at the other end of the reference.



```
package test;

class Animal
{
    public void sleep(){}
    public void makeNoise(){}
}

class Dog extends Animal
{
    public void roam(){}
    public void sleep(){}
    public void makeNoise(){}
}

class Cat{}

public class myTest
{
    public static void main(String[] args)
    {
        Animal a = new Dog();

        a.sleep();
        a.roam();
    }
}
```

variable a thinks that it should be an Animal object,
so it doesn't know the roam() method

- Imaging that someone want to reuse some methods in your Dog & Cat class in their PetShop program. But your Dog class do not have pet behaviors. (for example, play with his master)
- Think about what YOU would do if YOU were the Dog class programmer and needed to modify the Dog so that it could do Pet things, too.
- We know that simply adding new Pet behaviors (methods) to the Dog class will work, and won’t break anyone else’s code.
- Can you see any drawbacks to that approach (adding Pet methods to the Dog class)?
- just try to imagine how you’d like to solve the problem of modifying some of your Animal classes to include Pet behaviors.

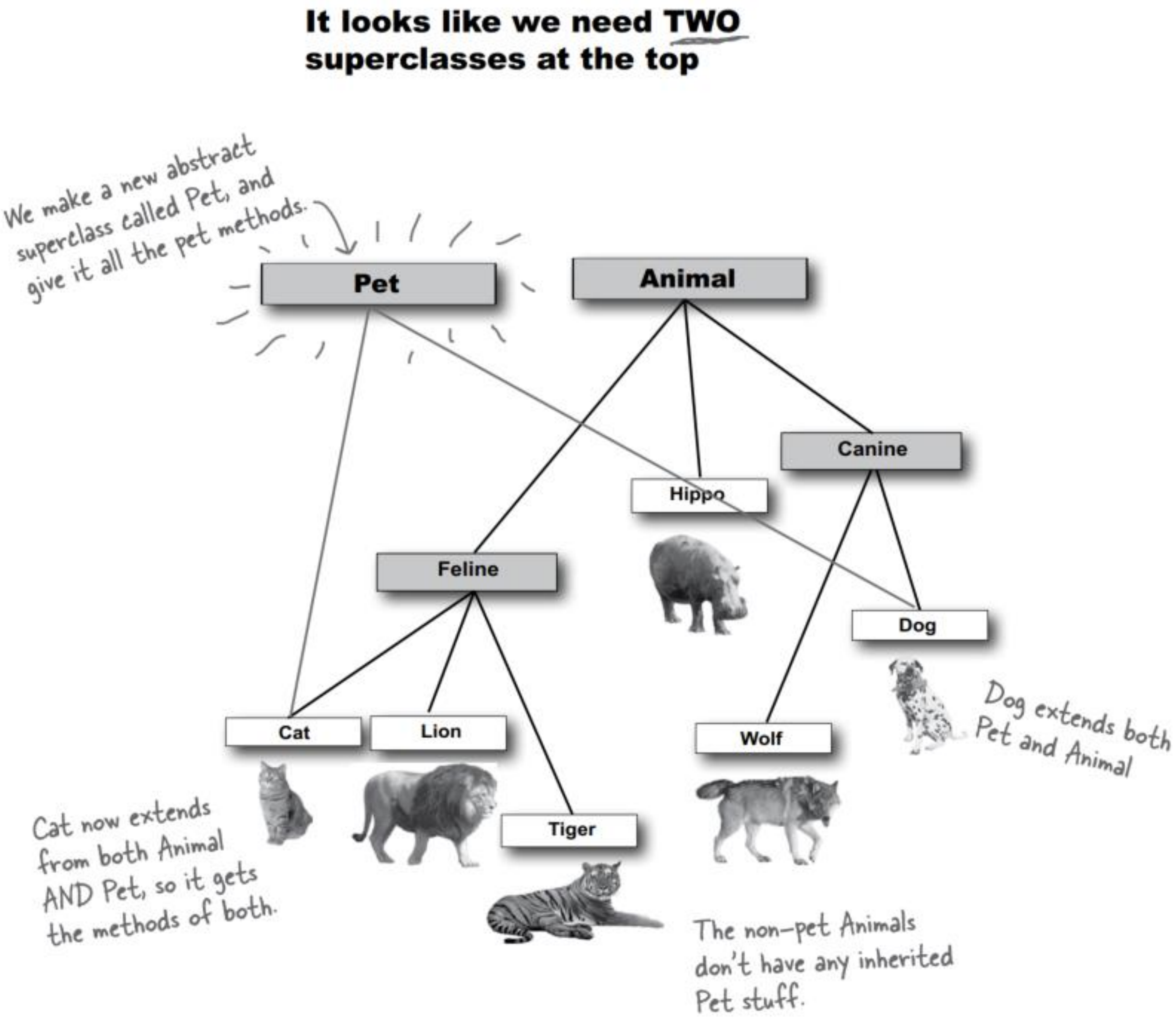
- Option 1: put pet methods in class Animal
- Pros: all the Animals will instantly inherit the pet behaviors. We won’t have to touch the existing Animal subclasses at all
- Cons: could be dangerous to give non-pets pet methods. Furthermore, we have to override these methods for many pet-like class (e.g. Dog, Cat, etc.) because they behave differently.

- Option 2: putting the pet methods in class Animal, but we make the methods abstract.
- Pros: all classes MUST override the methods, but they can make the methods do nothing.
- Cons: the concrete Animal subclasses are forced to implement all of them (even though the methods wouldn’t actually DO anything when called).

- Option 3: put the pet methods ONLY in the classes where they belong.
- Pros: Dogs can implement the methods and Cats can implement the methods, but nobody else has to know about them.
- Cons: Methods are implemented in concrete classes, so the compiler cannot check if the methods are inconsistent in two different classes. Besides, you don’t get to use polymorphism for the pet methods. In other words, you can’t use Animal as the polymorphic type now, because the compiler won’t let you call a Pet method on an Animal reference.

- What we REALLY need is:
- A way to have pet behavior in just the pet classes
- A way to guarantee that all pet classes have all of the same methods defined (same name, same arguments, same return types, no missing methods, etc.).
- A way to take advantage of polymorphism so that all pets can have their pet methods called, without having to use arguments, return types, and arrays for each and every pet class

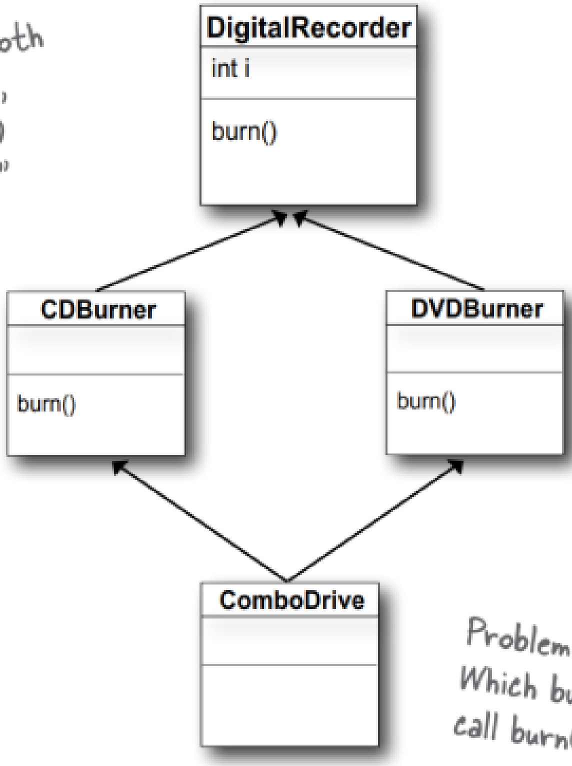
- Is there any elegant way to solve the problem?


It looks like we need TWO superclasses at the top
We make a new abstract superclass called Pet, and give it all the pet methods.
Pet
Animal
Canine
Hippo
Feline
Dog
Cat
Lion
Wolf
Tiger
Dog extends both Pet and Animal
Cat now extends from both Animal AND Pet, so it gets the methods of both.
The non-pet Animals don't have any inherited Pet stuff.

- Multiple inheritance is a bad thing.

Deadly Diamond of Death

CDBurner and DVDBurner both inherit from DigitalRecorder, and both override the burn() method. Both inherit the "i" instance variable.

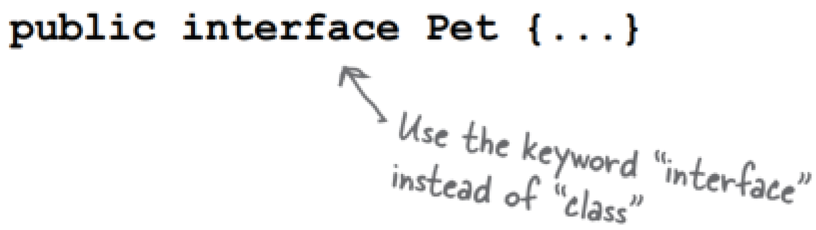


Imagine that the "i" instance variable is used by both CDBurner and DVDBurner, with different values. What happens if ComboDrive needs to use both values of "i"?

Problem with multiple inheritance. Which burn() method runs when you call burn() on the ComboDrive?

- *multiple inheritance is allowed in C++/Python. But this diamond problem still exists.

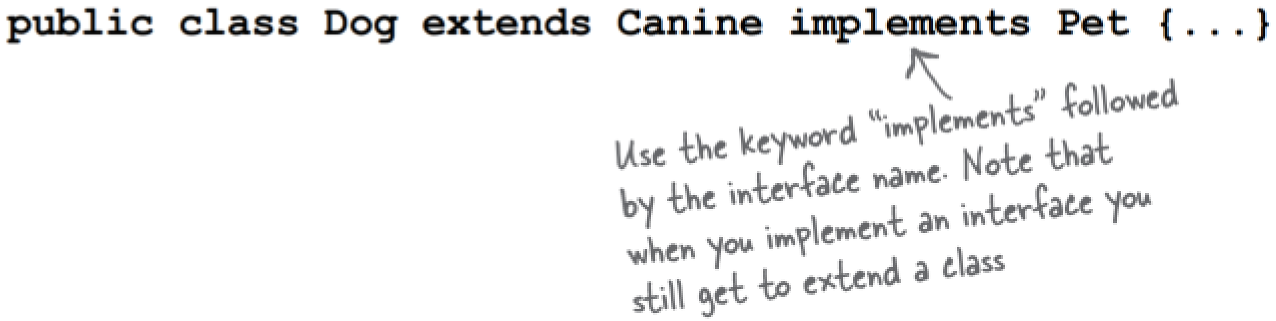
- Java gives you a solution. An ***interface***.
- A Java interface is like a 100% pure abstract class
- The subclass must implement the methods
- To define an interface:



```
public interface Pet {...}
```

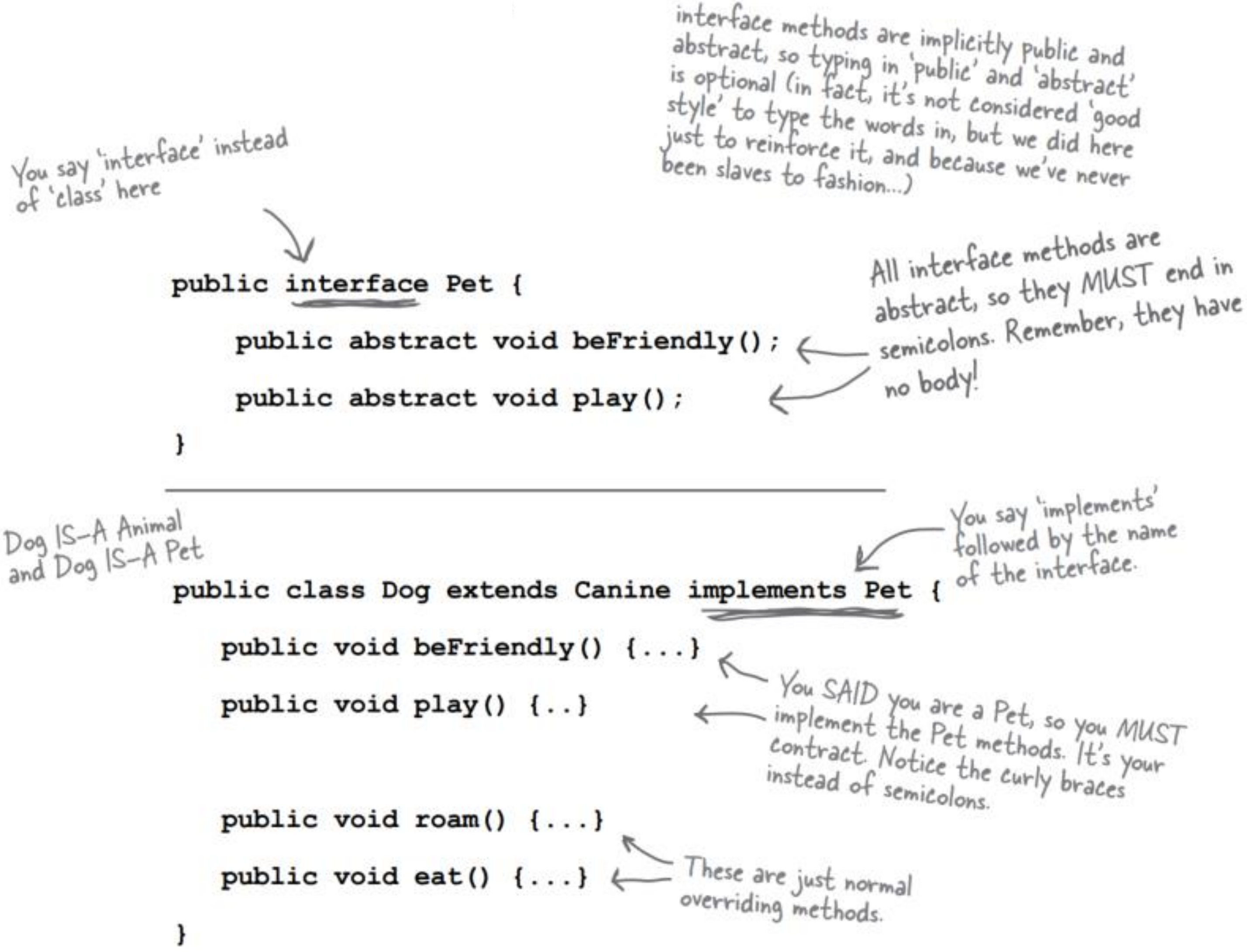
Use the keyword "interface" instead of "class"

- To implement an interface:



```
public class Dog extends Canine implements Pet {...}
```

Use the keyword "implements" followed by the interface name. Note that when you implement an interface you still get to extend a class



You say 'interface' instead of 'class' here

interface methods are implicitly public and abstract, so typing in 'public' and 'abstract' is optional (in fact, it's not considered 'good style' to type the words in, but we did here just to reinforce it, and because we've never been slaves to fashion...)

```
public interface Pet {

    public abstract void beFriendly();

    public abstract void play();

}
```

All interface methods are abstract, so they MUST end in semicolons. Remember, they have no body!

Dog IS-A Animal and Dog IS-A Pet

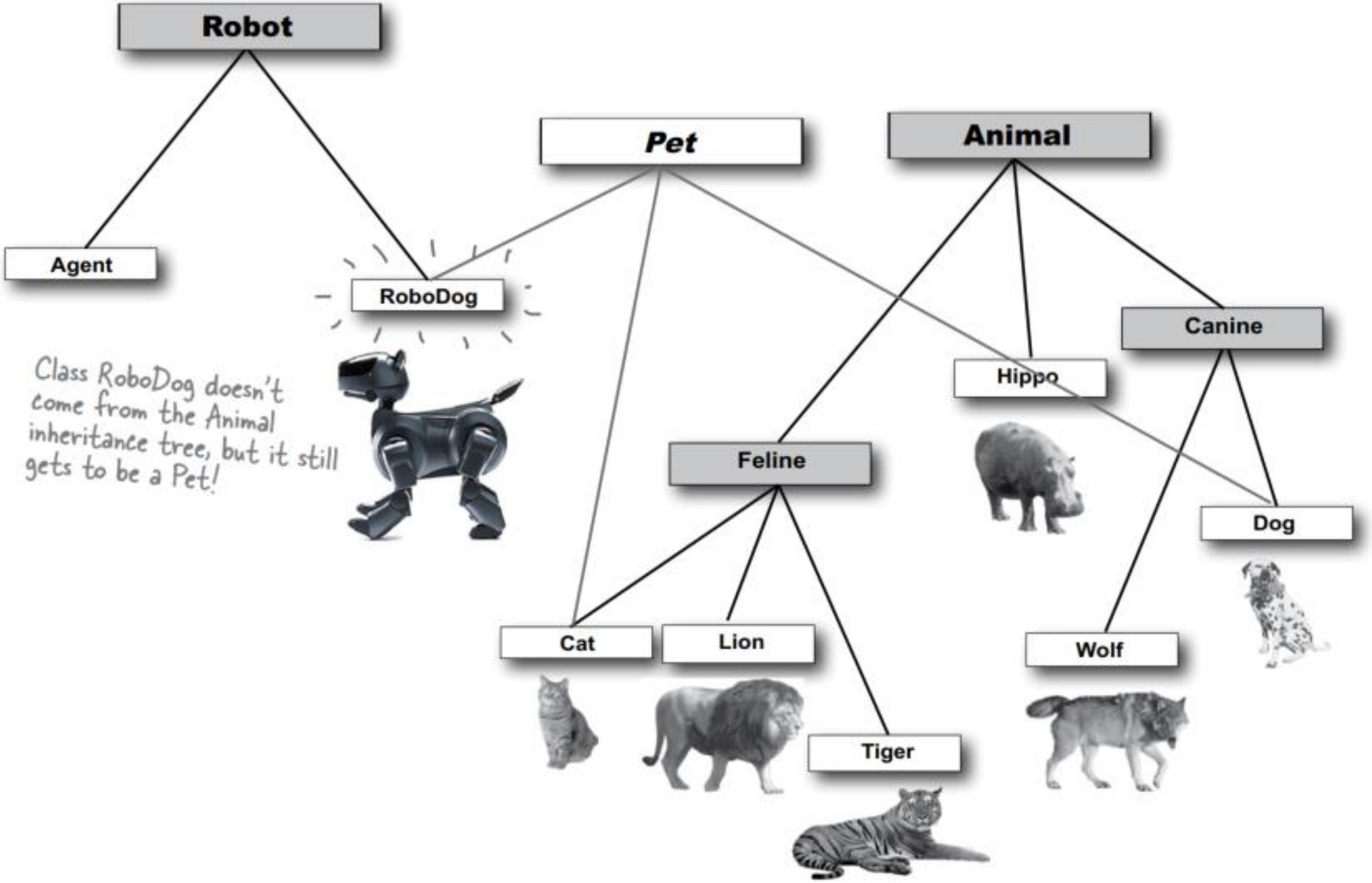
You say 'implements' followed by the name of the interface.

```
public class Dog extends Canine implements Pet {

   public void beFriendly() {...}

   public void play() {..}


   public void roam() {...}

   public void eat() {...}

}
```

You SAID you are a Pet, so you MUST implement the Pet methods. It's your contract. Notice the curly braces instead of semicolons.

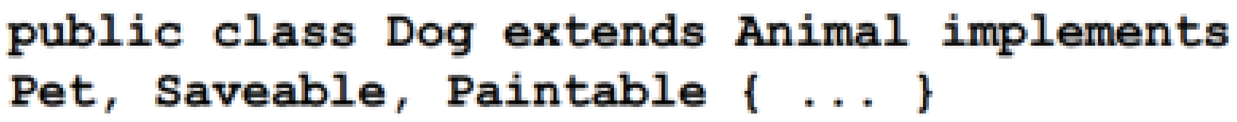
These are just normal overriding methods.

- Classes from different inheritance trees can implement the same interface



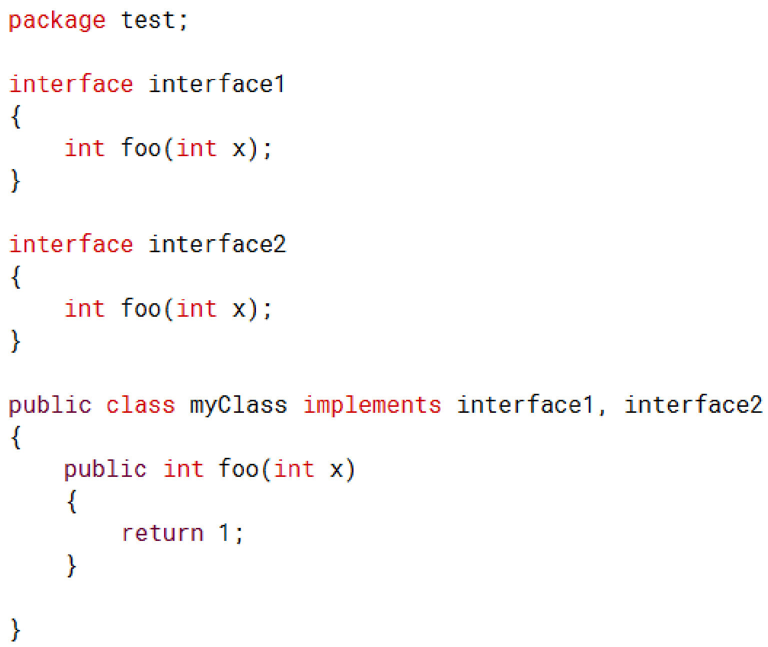
Class RoboDog doesn't come from the Animal inheritance tree, but it still gets to be a Pet!

- When you use a class as a polymorphic type (like an array of type Animal or a method that takes a Canine argument), the objects you can stick in that type must be from the same inheritance tree.
- But when you use an interface as a polymorphic type (like an array of Pets), the objects can be from anywhere in the inheritance tree. The only requirement is that the objects are from a class that implements the interface.
- If you want an object to be able to save its state to a file, just implement the Serializable interface.
- Better still, a class can implement multiple interfaces!

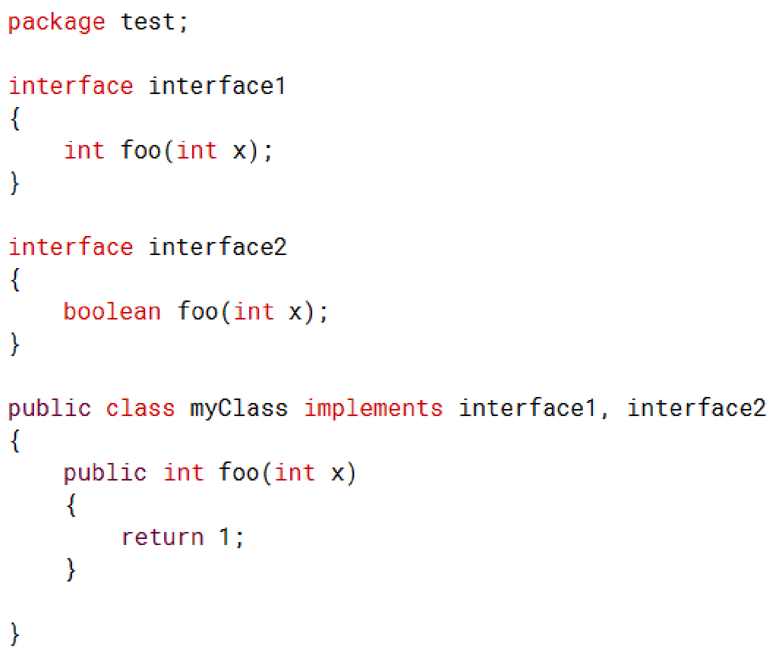


```
public class Dog extends Animal implements
Pet, Saveable, Paintable { ... }
```

- Quiz: what if two interfaces that have the same method?

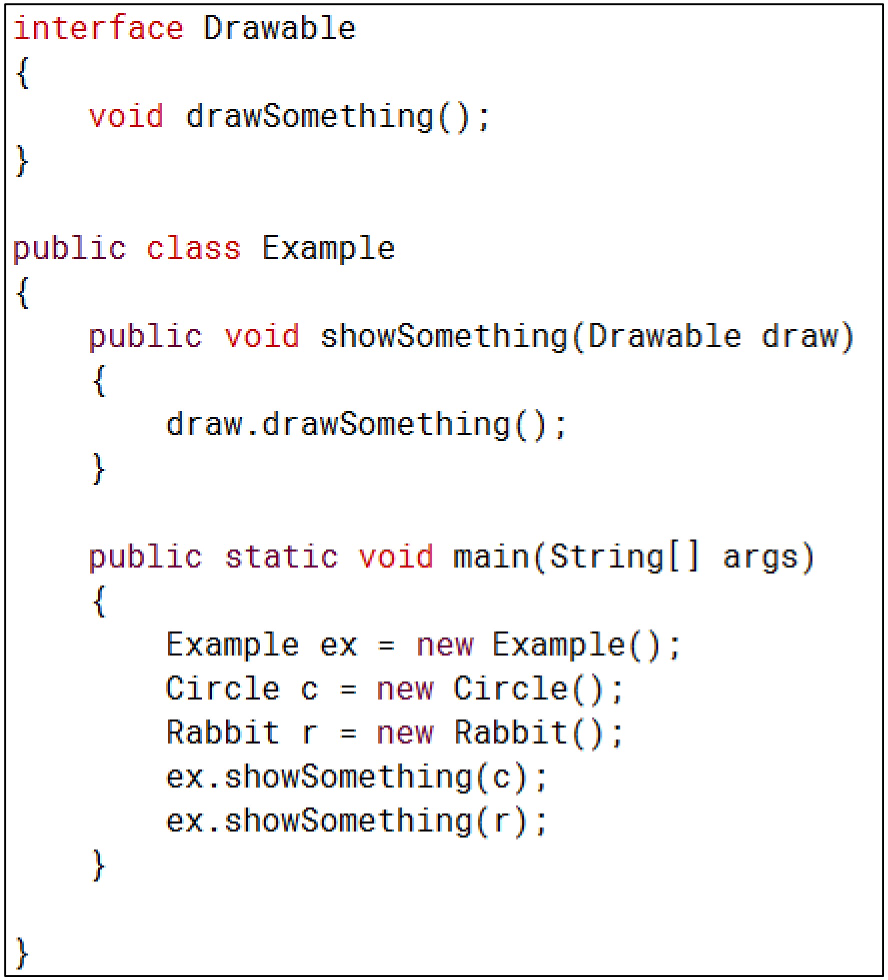


```java
package test;

interface interface1
{
    int foo(int x);
}

interface interface2
{
    int foo(int x);
}

public class myClass implements interface1, interface2
{
    public int foo(int x)
    {
        return 1;
    }

}
```



```java
package test;

interface interface1
{
    int foo(int x);
}

interface interface2
{
    boolean foo(int x);
}

public class myClass implements interface1, interface2
{
    public int foo(int x)
    {
        return 1;
    }

}
```

- Interface cannot be instantiate. It cannot have any instance variables. (But it can hold static & final variables. We will discuss that in chapter 10)
- Interface can be inherited. Use the same keyword ***extends***
- A very powerful thing: interface can be used as method parameters (the polymorphism!)



```
interface Drawable
{
    void drawSomething();
}

public class Example
{
    public void showSomething(Drawable draw)
    {
        draw.drawSomething();
    }

    public static void main(String[] args)
    {
        Example ex = new Example();
        Circle c = new Circle();
        Rabbit r = new Rabbit();
        ex.showSomething(c);
        ex.showSomething(r);
    }

}
```

- How do you know whether to make a class, a subclass, an abstract class, or an interface?
- Make a class that doesn’t extend anything (other than Object) when your new class doesn’t pass the IS-A test for any other type.
- Make a subclass (in other words, extend a class) only when you need to make a more specific version of a class and need to override or add new behaviors.
- Use an abstract class when you want to define a template for a group of subclasses, and you have at least some implementation code that all subclasses could use. Make the class abstract when you want to guarantee that nobody can make objects of that type.
- Use an interface when you want to define a role that other classes can play, regardless of where those classes are in the inheritance tree.

- Several typical interfaces in Java
- Runnable, Callable – used by any class whose instances are intended to be executed by a thread
- Serializable – for I/O purpose
- List – used by Collections (e.g. ArrayList)
- Iterable - allows an object to be the target of the "for-each loop" statement
- There are many other interfaces in Java!

- Summary
- About abstract class
- When you don’t want a class to be instantiated (in other words, you don’t want anyone to make a new object of that class type) mark the class with the ***abstract*** keyword
- If a class has even ONE abstract method, the class must be marked abstract
- An abstract class can have both abstract and non-abstract methods
- An abstract method has no body, and the declaration ends with a semicolon (no curly braces)

- About Object class
- Every class in Java is either a direct or indirect subclass of class Object (java.lang.Object)
- Methods can be declared with Object arguments and/or return types
- You can call methods on an object only if the methods are in the class (or interface) used as the reference variable type, regardless of the actual object type. So, a reference variable of type Object can be used only to call methods defined in class Object, regardless of the type of the object to which the reference refers.
- A reference variable of type Object can’t be assigned to any other reference type without a cast. A cast can be used to assign a reference variable of one type to a reference variable of a subtype, but at runtime the cast will fail if the object on the heap is NOT of a type compatible with the cast
- All objects come out of an ArrayList<Object> as type Object (meaning, they can be referenced only by an Object reference variable, unless you use a cast).

- About interface
- An interface is like a 100% pure abstract class. It defines *only* abstract methods
- Create an interface using the interface keyword instead of the word class
- Your class can implement multiple interfaces
- A class that implements an interface must implement all the methods of the interface, since all interface methods are implicitly public and abstract

- Example of polymorphism and interface
- Assume that you have two shapes, a rectangle and a circle. You are planning to have more shapes such as polygons or irregular shapes in the future.
- Now, you have to implement a method which is going to calculate the sum of any two "things". What will be your design?

- Solution
- Create an interface named Measurable. Make an abstract method called measureArea().
- Create several classes such as Circle, Rectangle, etc. Let them implements the Measurable interface.
- In another class (which is describing your main logic), create a method called sumArea(), make the argument type as Measurable.

Given:

1)
```
public interface Foo { }
public class Bar implements Foo { }
```

2)
```
public interface Vinn { }
public abstract class Vout implements Vinn { }
```

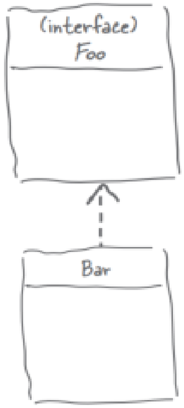
3)
```
public abstract class Muffie implements Whuffie { }
public class Fluffie extends Muffie { }
public interface Whuffie { }
```

4)
```
public class Zoop { }
public class Boop extends Zoop { }
public class Goop extends Boop { }
```

5)
```
public class Gamma extends Delta implements Epsilon { }
public interface Epsilon { }
public interface Beta { }
public class Alpha extends Gamma implements Beta { }
public class Delta { }
```

What's the Picture ?

1)



2)

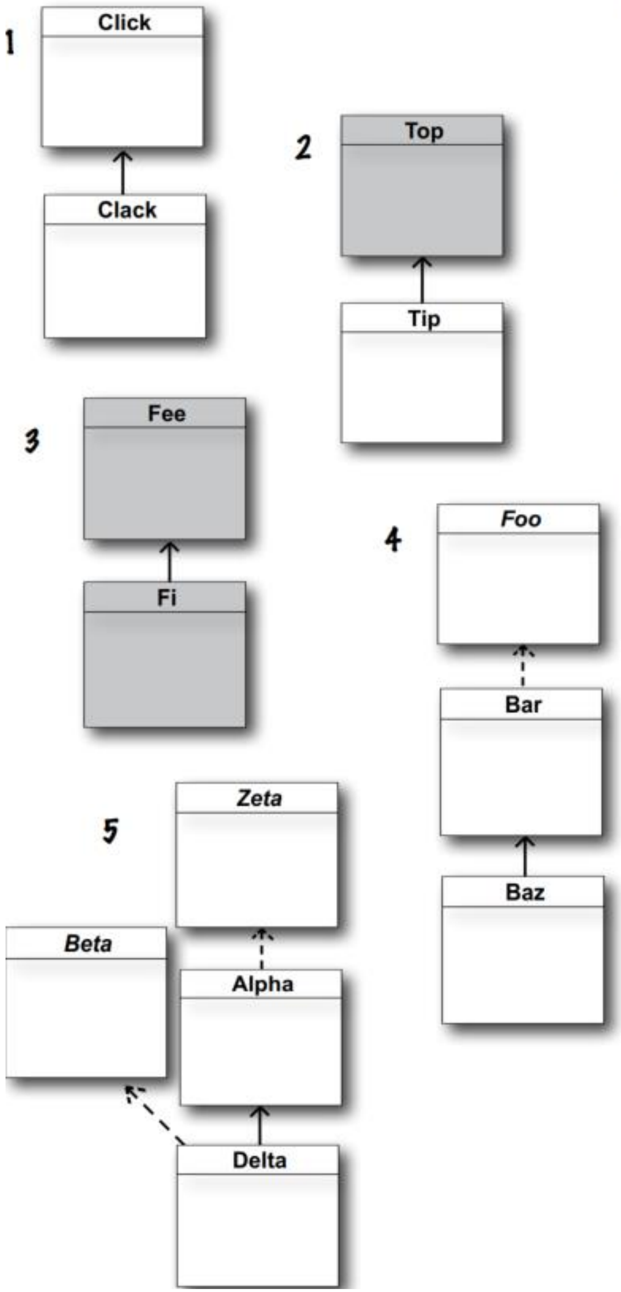
3)

4)

5)

What's the Declaration ?

Given:



1) public class Click { }

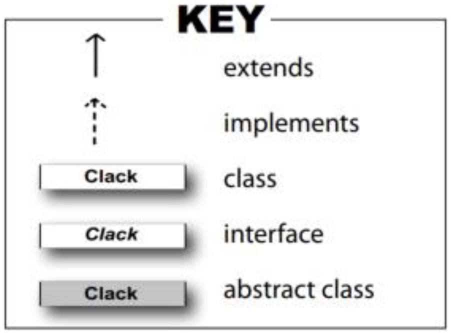
public class Clack extends Click { }

2)

3)

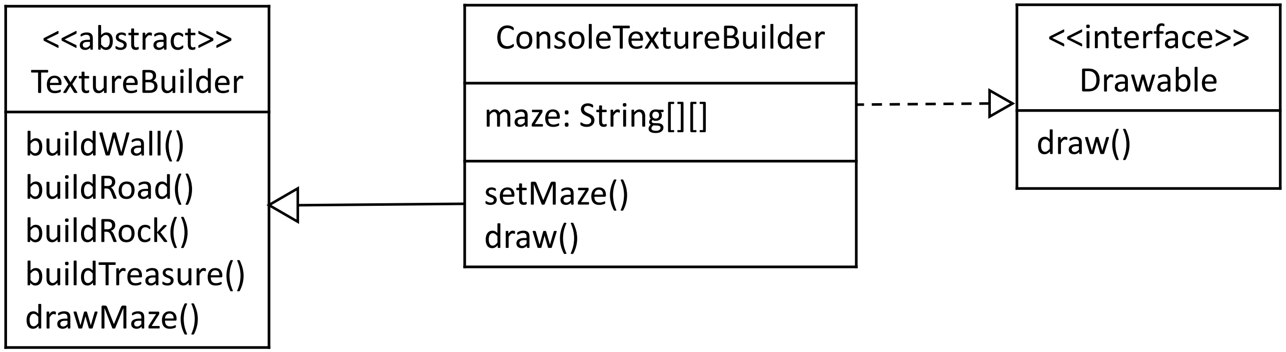
4)

5)



- Exercise: Suppose that you want to design a maze game. First of all, you have to create a maze.
- Think about how to elegantly design the maze structure. For example, you should separate the conceptual maze structure from the display UI.

- Create an abstract class (or interface) which defines what you want to build in the maze. Create a concrete builder class (For example, ConsoleTextureBuilder) to implements these methods



<<abstract>>
TextureBuilder

buildWall()
buildRoad()
buildRock()
buildTreasure()
drawMaze()

ConsoleTextureBuilder

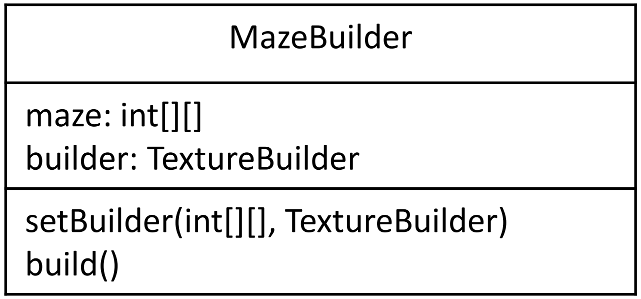
maze: String[][]

setMaze()
draw()

<<interface>>
Drawable

draw()

- Create a class which build the logical maze. This class has a maze data and the texture it uses.

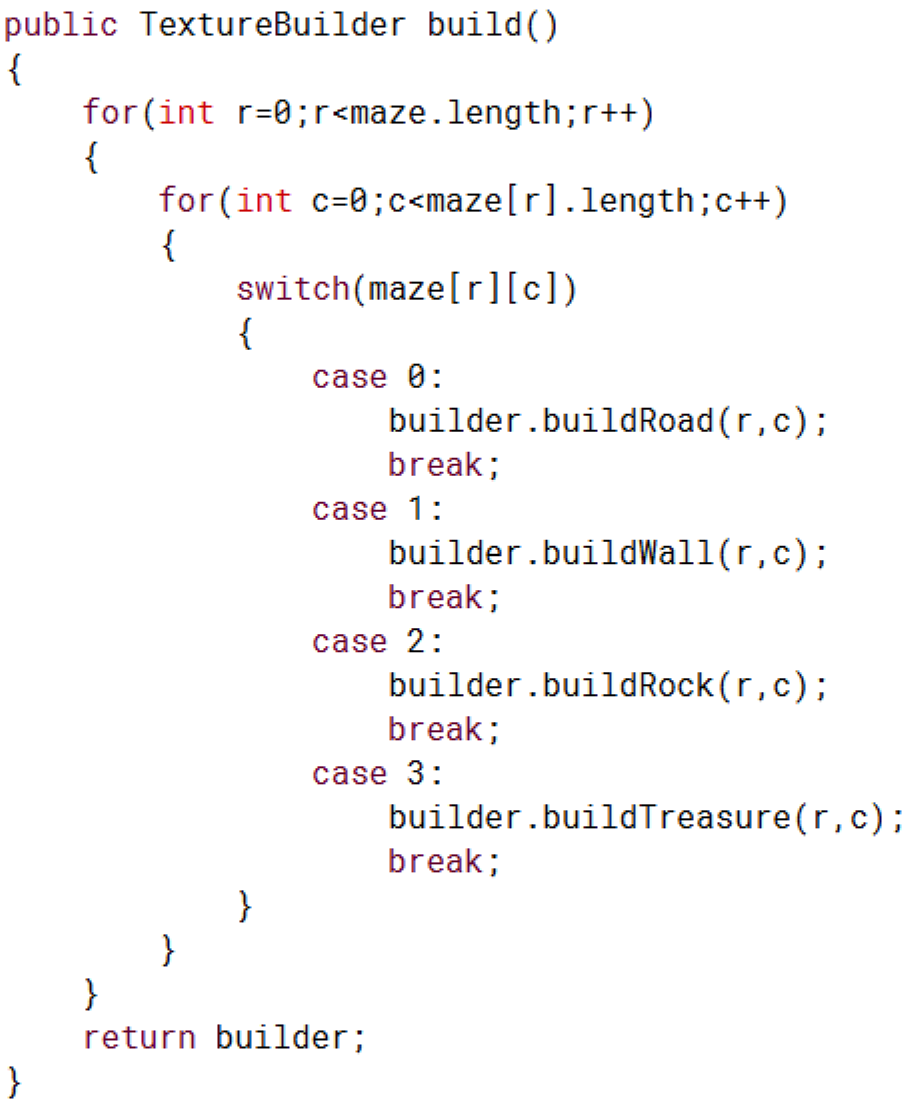


MazeBuilder

maze: int[][]
builder: TextureBuilder

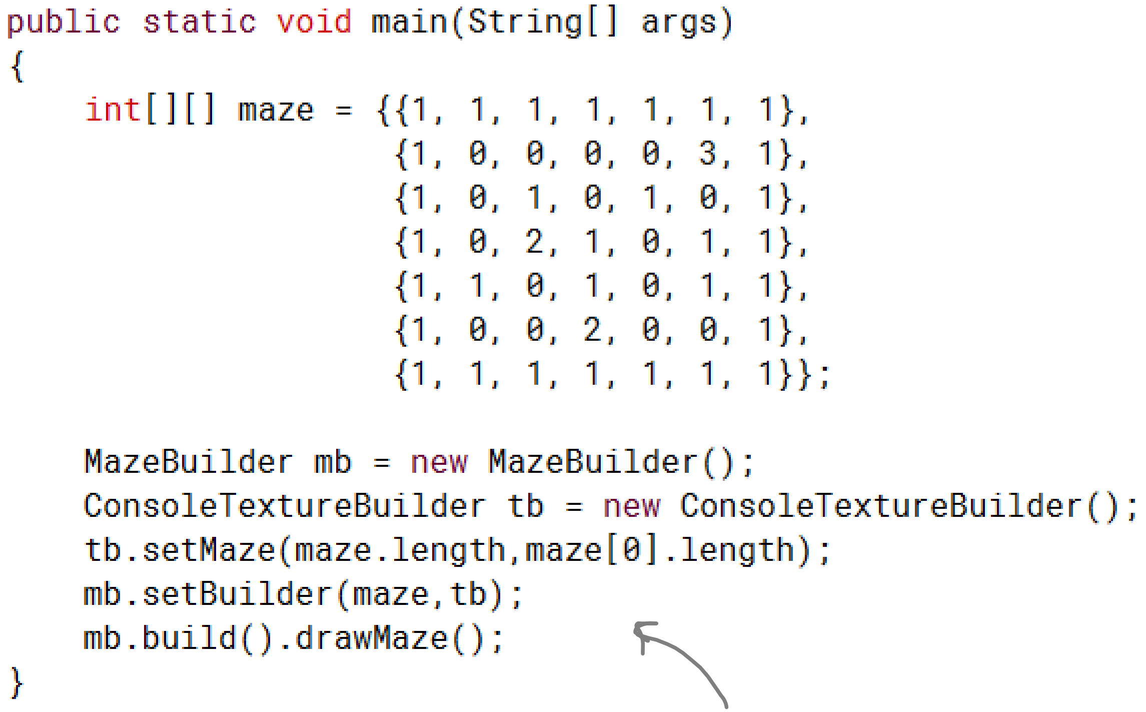
setBuilder(int[][], TextureBuilder)
build()

- Example of the build() method



```
public TextureBuilder build()
{
    for(int r=0;r<maze.length;r++)
    {
        for(int c=0;c<maze[r].length;c++)
        {
            switch(maze[r][c])
            {
                case 0:
                    builder.buildRoad(r,c);
                    break;
                case 1:
                    builder.buildWall(r,c);
                    break;
                case 2:
                    builder.buildRock(r,c);
                    break;
                case 3:
                    builder.buildTreasure(r,c);
                    break;
            }
        }
    }
    return builder;
}
```

- Example of main class



```
public static void main(String[] args)
{
    int[][] maze = {{1, 1, 1, 1, 1, 1, 1},
                    {1, 0, 0, 0, 0, 3, 1},
                    {1, 0, 1, 0, 1, 0, 1},
                    {1, 0, 2, 1, 0, 1, 1},
                    {1, 1, 0, 1, 0, 1, 1},
                    {1, 0, 0, 2, 0, 0, 1},
                    {1, 1, 1, 1, 1, 1, 1}};

    MazeBuilder mb = new MazeBuilder();
    ConsoleTextureBuilder tb = new ConsoleTextureBuilder();
    tb.setMaze(maze.length,maze[0].length);
    mb.setBuilder(maze,tb);
    mb.build().drawMaze();
}
```

You may not have to use setter here if you define constructors