Head First

Chapter-8: Interfaces and Abstract classes

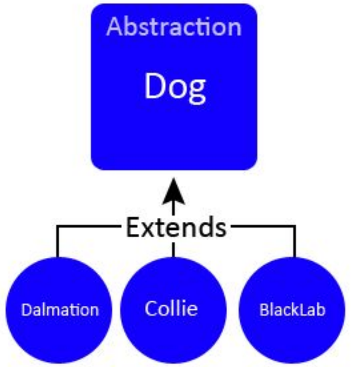
Upcode Software
Engineer Team

-2023-

CONTENT

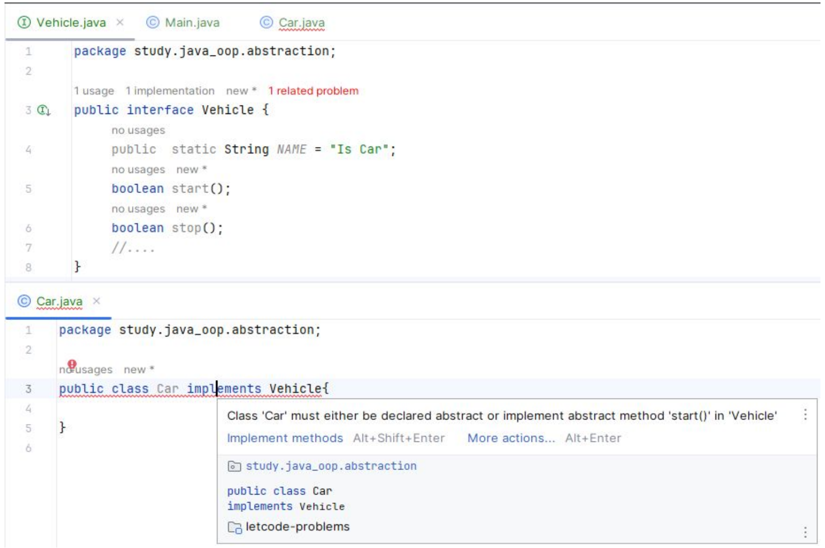
What is an abstract class

- *It's a class that can't be instantiated*
- The compiler won't let you instantiate an abstract class
- You don't have to worry about somebody making objects of that type. The compiler guarantees it.
- An abstract class has virtually* no use, no value, no purpose in life, unless it is extended.



What is an interface

- It's a **100% *abstract*** class
- **Interfaces** are the **poly** in **polymorphism**. (**poly** = many **morphism** = behavior or form)



Vehicle.java × Main.java Car.java

```
package study.java_oop.abstraction;

1 usage   1 implementation   new *   1 related problem
public interface Vehicle {
        no usages
        public  static String NAME = "Is Car";
        no usages   new *
        boolean start();
        no usages   new *
        boolean stop();
        //....
}
```

Car.java ×

```
package study.java_oop.abstraction;

no usages   new *
public class Car implements Vehicle{

}
```

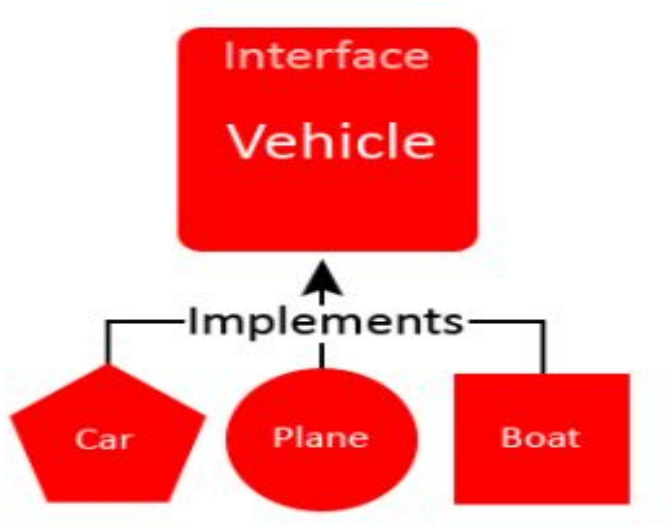
Class 'Car' must either be declared abstract or implement abstract method 'start()' in 'Vehicle'

Implement methods Alt+Shift+Enter More actions... Alt+Enter

study.java_oop.abstraction

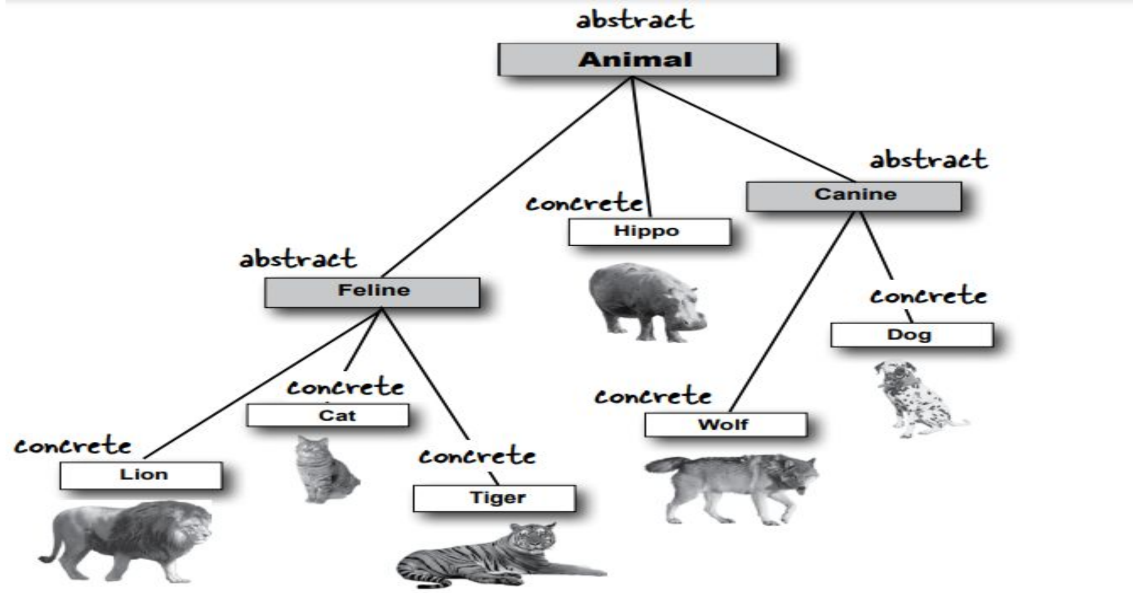
```
public class Car
implements Vehicle
```

letcode-problems



Why/Where do we need to abstract class?

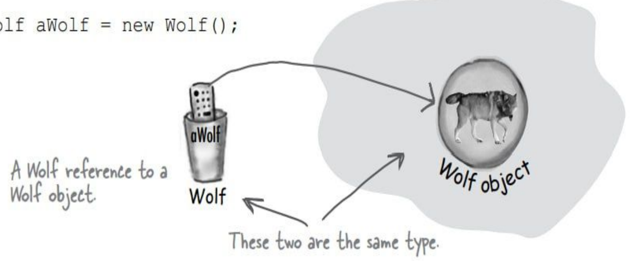
- *We need to go beyond simple inheritance to a level*
- Flexibility and extensibility you can get only by designing and coding to interface specifications



Where do we use it?

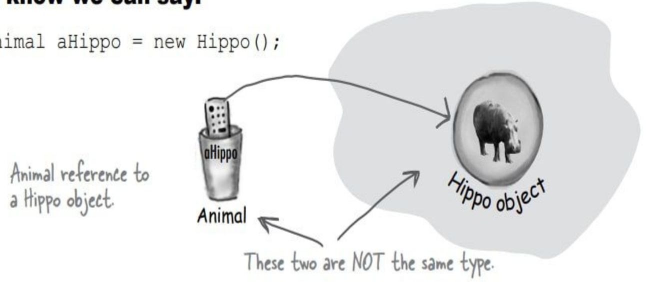
We know we can say:

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


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

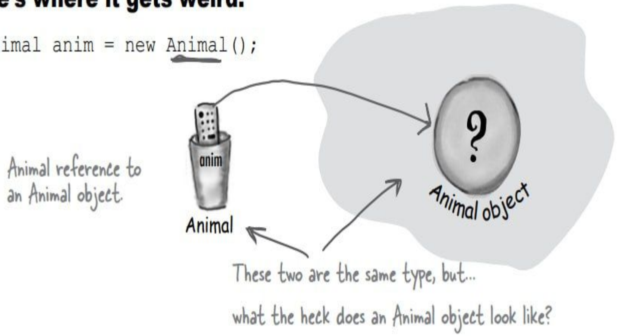
And we know we can say:

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

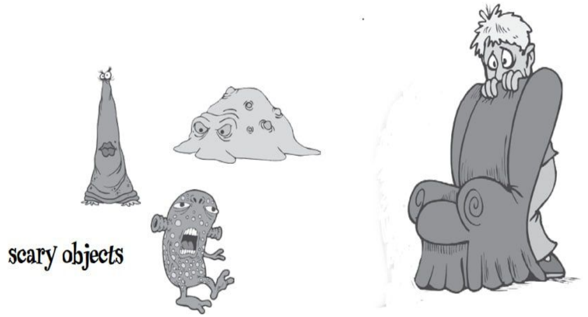

aHippo
Animal
Hippo object
Animal reference to a Hippo object.
These two are NOT the same type.

But here's where it gets weird:

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

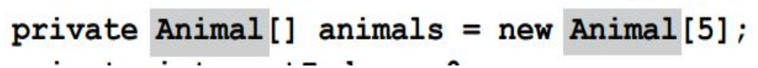

anim
Animal
?
Animal object
Animal reference to an Animal object.
These two are the same type, but...
what the heck does an Animal object look like?

What does a new Animal() object *look* like?

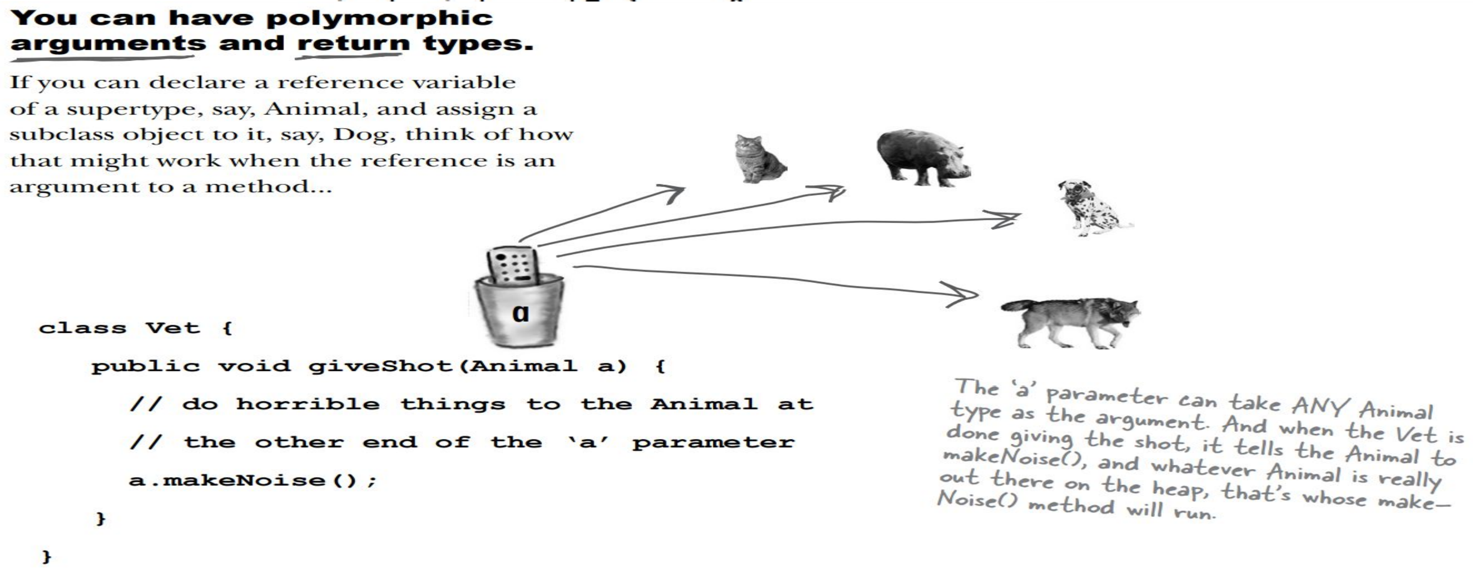


Where do we use it?

Polymorphism! Remember, what you want is the ability to use a superclass type (often abstract) as a method argument, return type, or array type.



```
private Animal[] animals = new Animal[5];
```



You can have polymorphic arguments and return types.

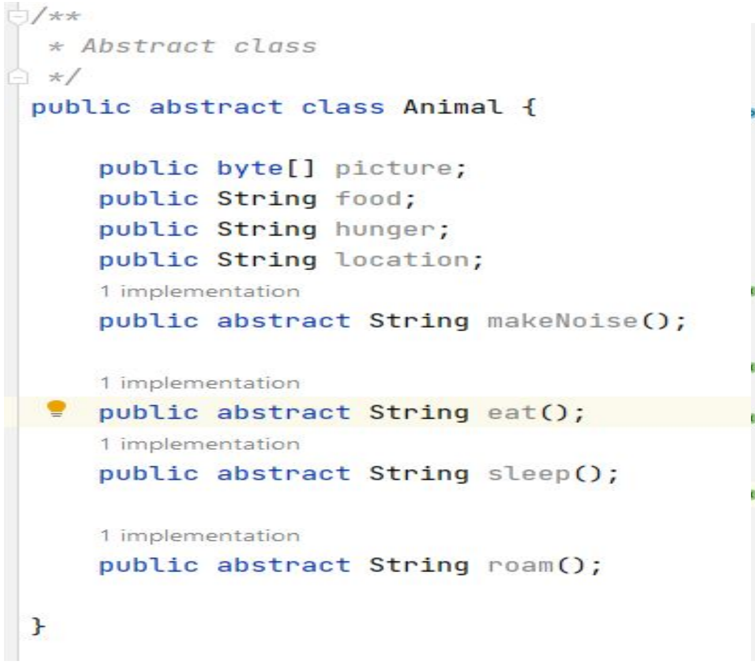
If you can declare a reference variable of a supertype, say, Animal, and assign a subclass object to it, say, Dog, think of how that might work when the reference is an argument to a method...

```
class Vet {

   public void giveShot(Animal a)  {

      // do horrible things to the Animal at

      // the other end of the 'a' parameter

      a.makeNoise();

   }

}
```

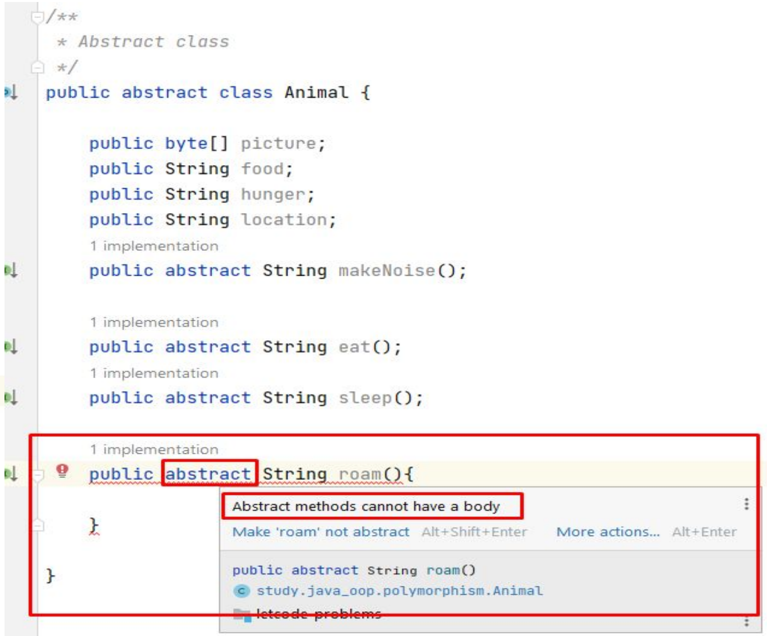
The 'a' parameter can take ANY Animal type as the argument. And when the Vet is done giving the shot, it tells the Animal to makeNoise(), and whatever Animal is really out there on the heap, that's whose make-Noise() method will run.

Abstract method.

1. Abstract methods don’t have a body; they exist solely for polymorphism
2. An abstract class means the class must be extended; an abstract method



```
/**
 * Abstract class
 */
public abstract class Animal {

    public byte[] picture;
    public String food;
    public String hunger;
    public String location;
    1 implementation
    public abstract String makeNoise();

    1 implementation
    public abstract String eat();
    1 implementation
    public abstract String sleep();

    1 implementation
    public abstract String roam();

}
```



```
/**
 * Abstract class
 */
public abstract class Animal {

    public byte[] picture;
    public String food;
    public String hunger;
    public String location;
    1 implementation
    public abstract String makeNoise();

    1 implementation
    public abstract String eat();
    1 implementation
    public abstract String sleep();

    1 implementation
    public abstract String roam(){

    }

}
```

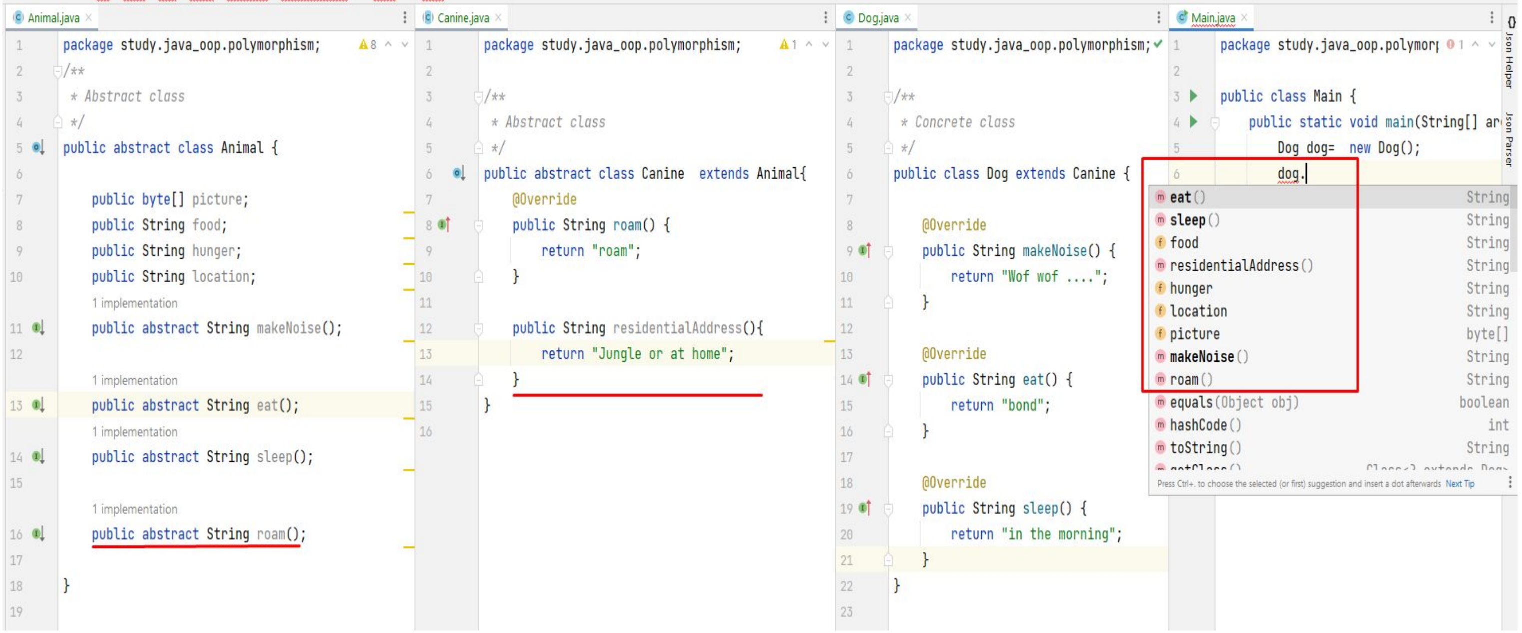
Abstract methods cannot have a body

Make 'roam' not abstract Alt+Shift+Enter   More actions... Alt+Enter

```
public abstract String roam()
study.java_oop.polymorphism.Animal
```

~~letcode-problems~~

Why/Where do we need to abstract class?



Animal.java

```java
package study.java_oop.polymorphism;
/**
 * Abstract class
 */
public abstract class Animal {

    public byte[] picture;
    public String food;
    public String hunger;
    public String location;
    1 implementation
    public abstract String makeNoise();

    1 implementation
    public abstract String eat();
    1 implementation
    public abstract String sleep();

    1 implementation
    public abstract String roam();

}
```

Canine.java

```java
package study.java_oop.polymorphism;

/**
 * Abstract class
 */
public abstract class Canine  extends Animal{
    @Override
    public String roam() {
        return "roam";
    }

    public String residentialAddress(){
        return "Jungle or at home";
    }
}
```

Dog.java

```java
package study.java_oop.polymorphism;

/**
 * Concrete class
 */
public class Dog extends Canine {

    @Override
    public String makeNoise() {
        return "Wof wof ....";
    }

    @Override
    public String eat() {
        return "bond";
    }

    @Override
    public String sleep() {
        return "in the morning";
    }
}
```

Main.java

```java
package study.java_oop.polymor

public class Main {
    public static void main(String[] ar
        Dog dog=  new Dog();
        dog.
```

```
eat()                      String
sleep()                    String
food                       String
residentialAddress()       String
hunger                     String
location                   String
picture                    byte[]
makeNoise()                String
roam()                     String
equals(Object obj)         boolean
hashCode()                 int
toString()                 String
Press Ctrl+. to choose the selected (or first) suggestion and insert a dot afterwards  Next Tip
```

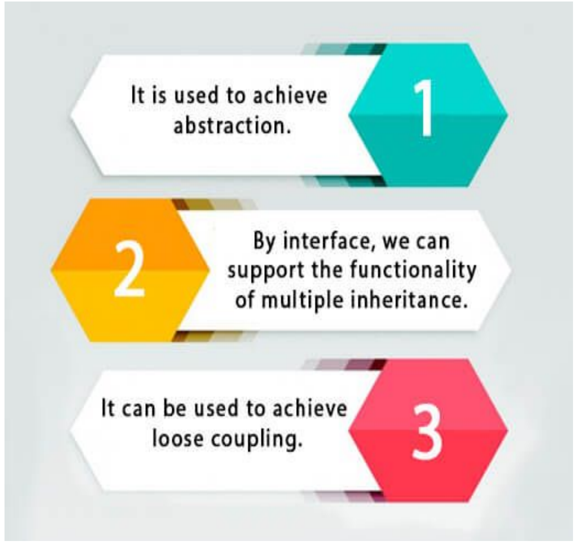
You can't think of any generic method implementation that could possibly be useful for subclasses.

Why/where do we need an interface?

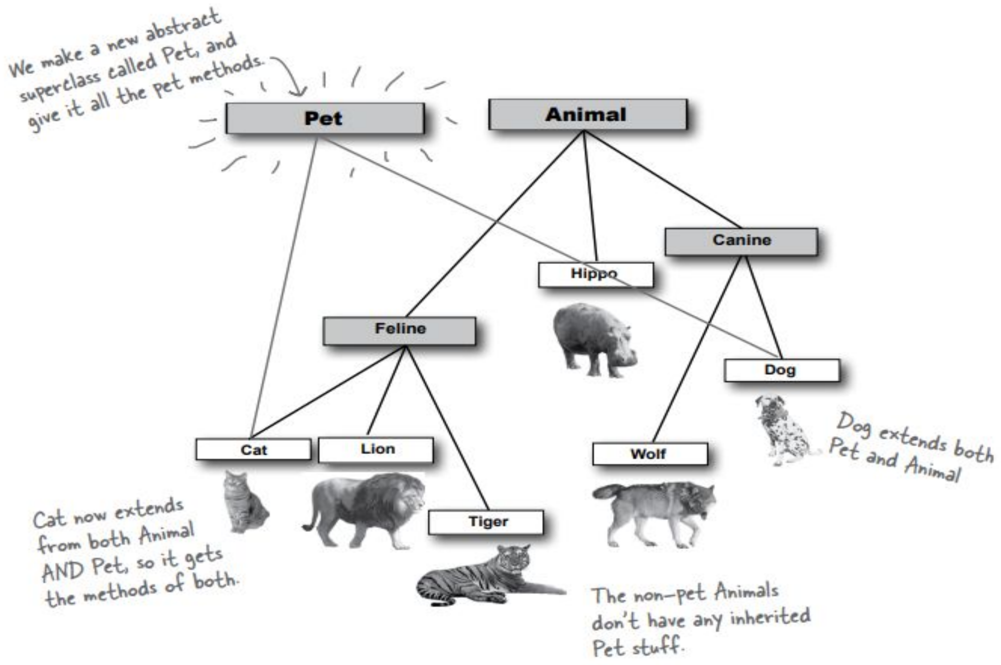
Why do we need an interface

1. In Java, an interface specifies the behavior of a class by providing an abstract type.
2. As one of Java's core concepts, abstraction, polymorphism, and multiple inheritance are supported through this technology.
3. Interfaces are used in Java to achieve abstraction.

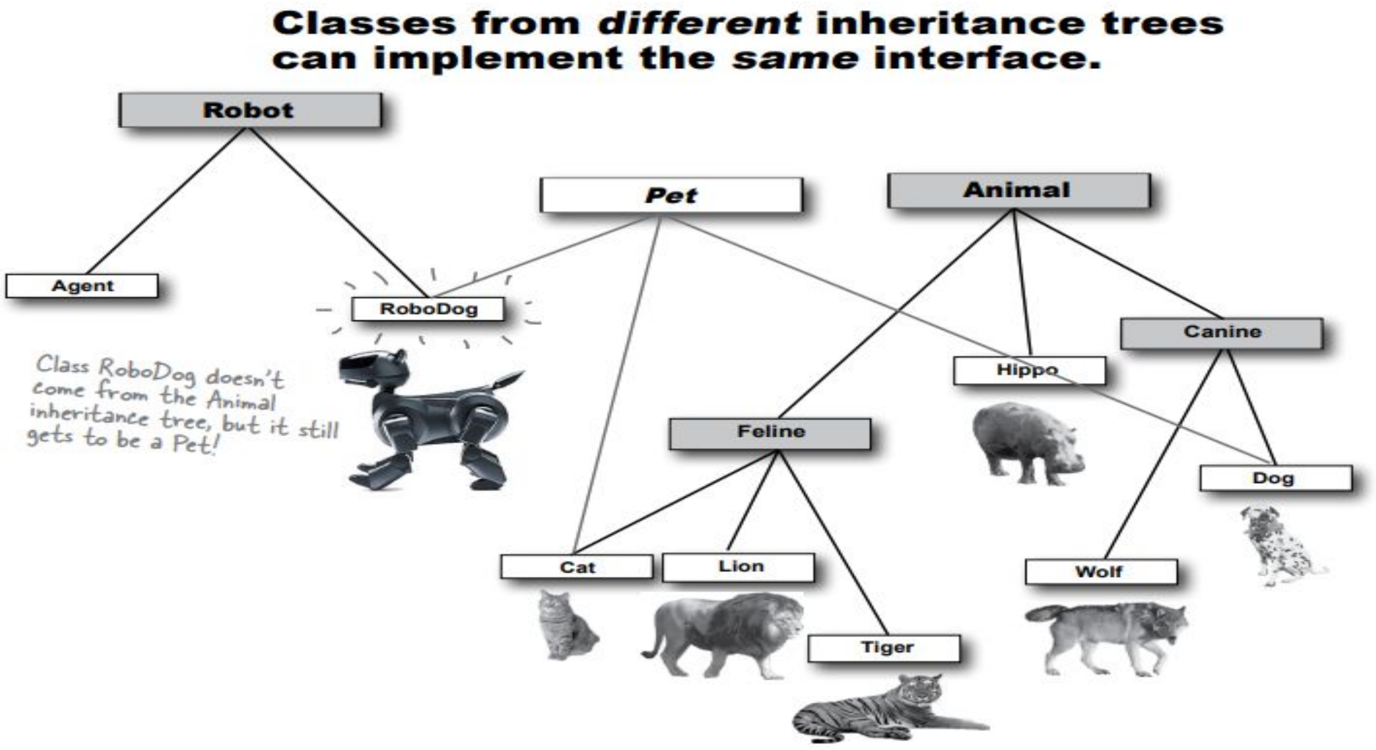
Need for Interface in Java


1
It is used to achieve abstraction.
2
By interface, we can support the functionality of multiple inheritance.
3
It can be used to achieve loose coupling.

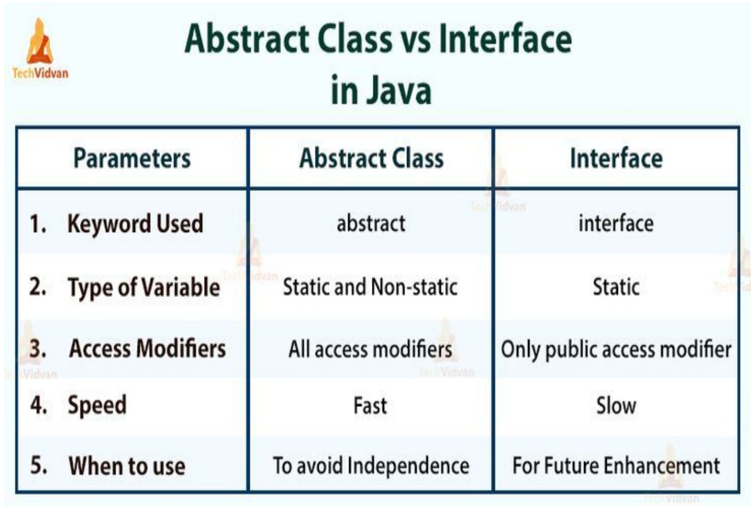
Why/where do we need an interface?


We make a new abstract superclass called Pet, and give it all the pet methods.
Pet
Animal
Canine
Hippo
Feline
Dog
Cat
Lion
Tiger
Wolf
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.

Why do we need an interface?

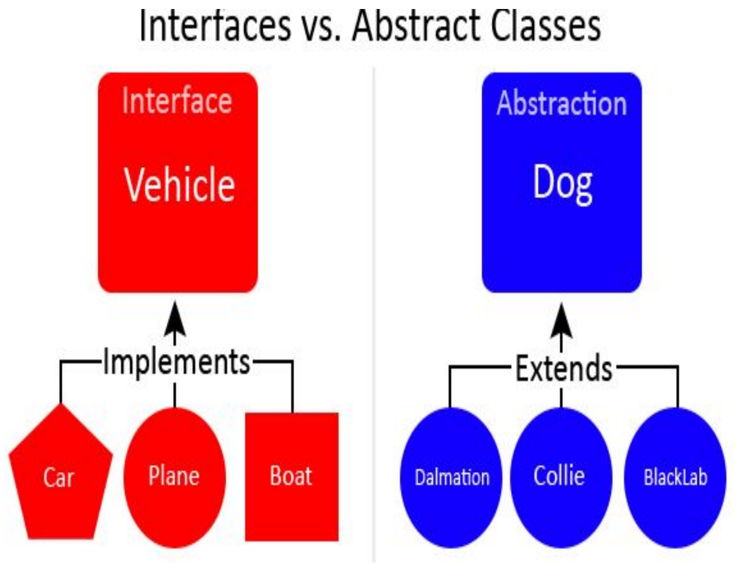

Classes from *different* inheritance trees can implement the *same* interface.
Robot
Pet
Animal
Agent
RoboDog
Canine
Hippo
Feline
Dog
Cat
Lion
Tiger
Wolf
Class RoboDog doesn't come from the Animal inheritance tree, but it still gets to be a Pet!

Abstract class and Interface

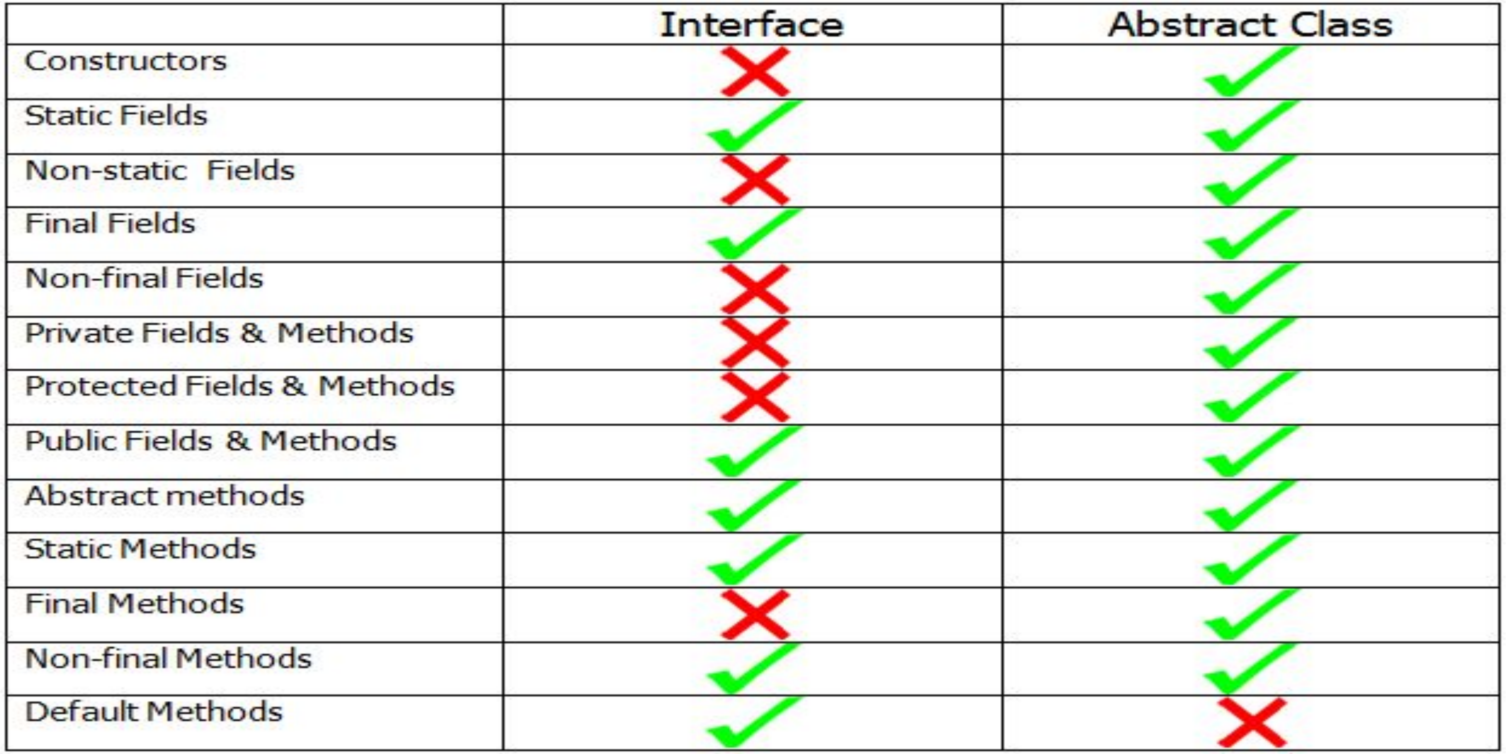


Abstract Class vs Interface in Java

Parameters	Abstract Class	Interface
1. Keyword Used	abstract	interface
2. Type of Variable	Static and Non-static	Static
3. Access Modifiers	All access modifiers	Only public access modifier
4. Speed	Fast	Slow
5. When to use	To avoid Independence	For Future Enhancement

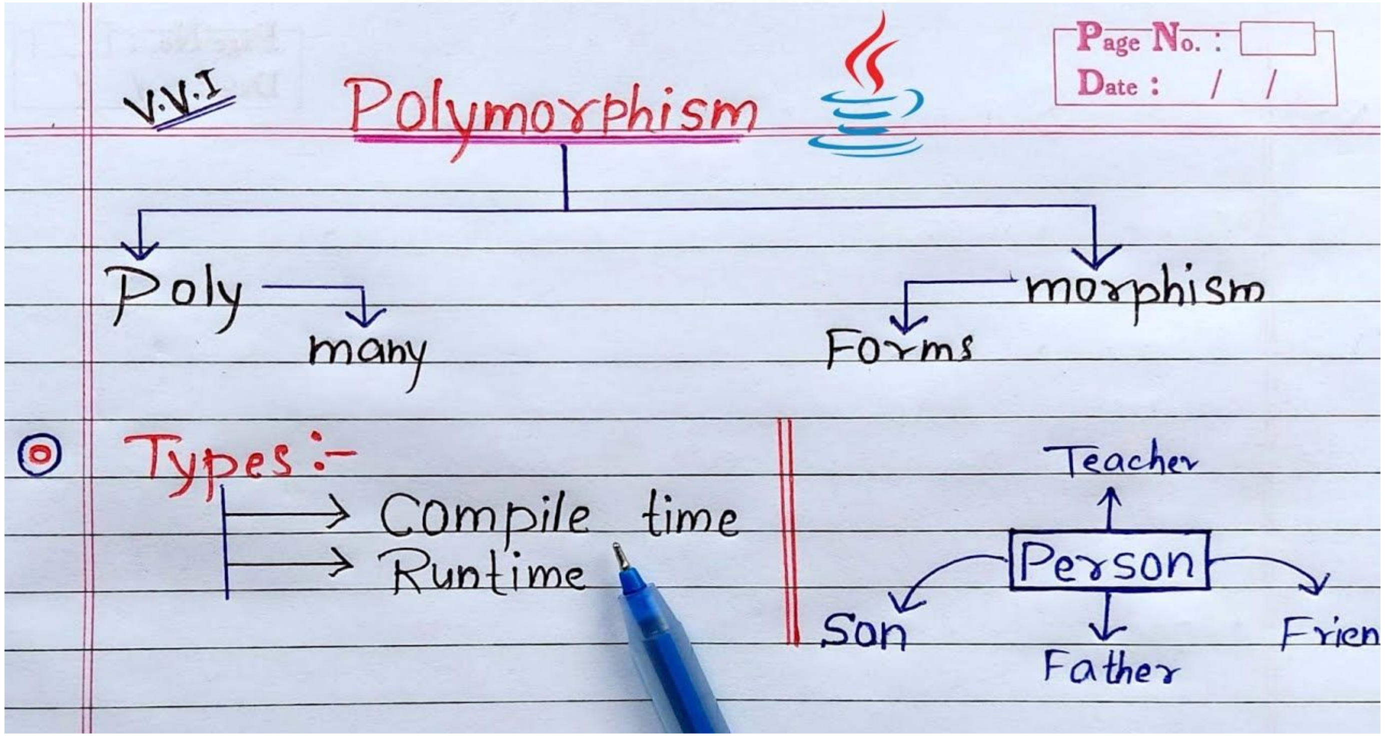


Abstract class and Interface

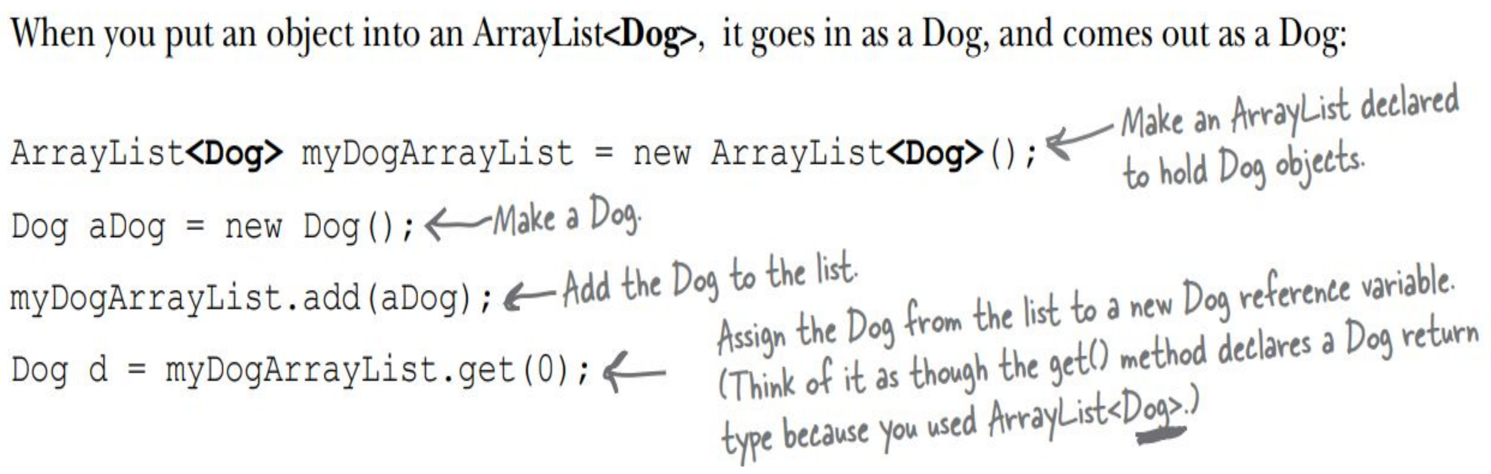


	Interface	Abstract Class
Constructors	✗	✓
Static Fields	✓	✓
Non-static Fields	✗	✓
Final Fields	✓	✓
Non-final Fields	✗	✓
Private Fields & Methods	✗	✓
Protected Fields & Methods	✗	✓
Public Fields & Methods	✓	✓
Abstract methods	✓	✓
Static Methods	✓	✓
Final Methods	✗	✓
Non-final Methods	✓	✓
Default Methods	✓	✗

Polymorphism



Using polymorphic references of type Object



When you put an object into an ArrayList<**Dog**>, it goes in as a Dog, and comes out as a Dog:

```
ArrayList<Dog> myDogArrayList = new ArrayList<Dog>();
Dog aDog = new Dog();
myDogArrayList.add(aDog);
Dog d = myDogArrayList.get(0);
```

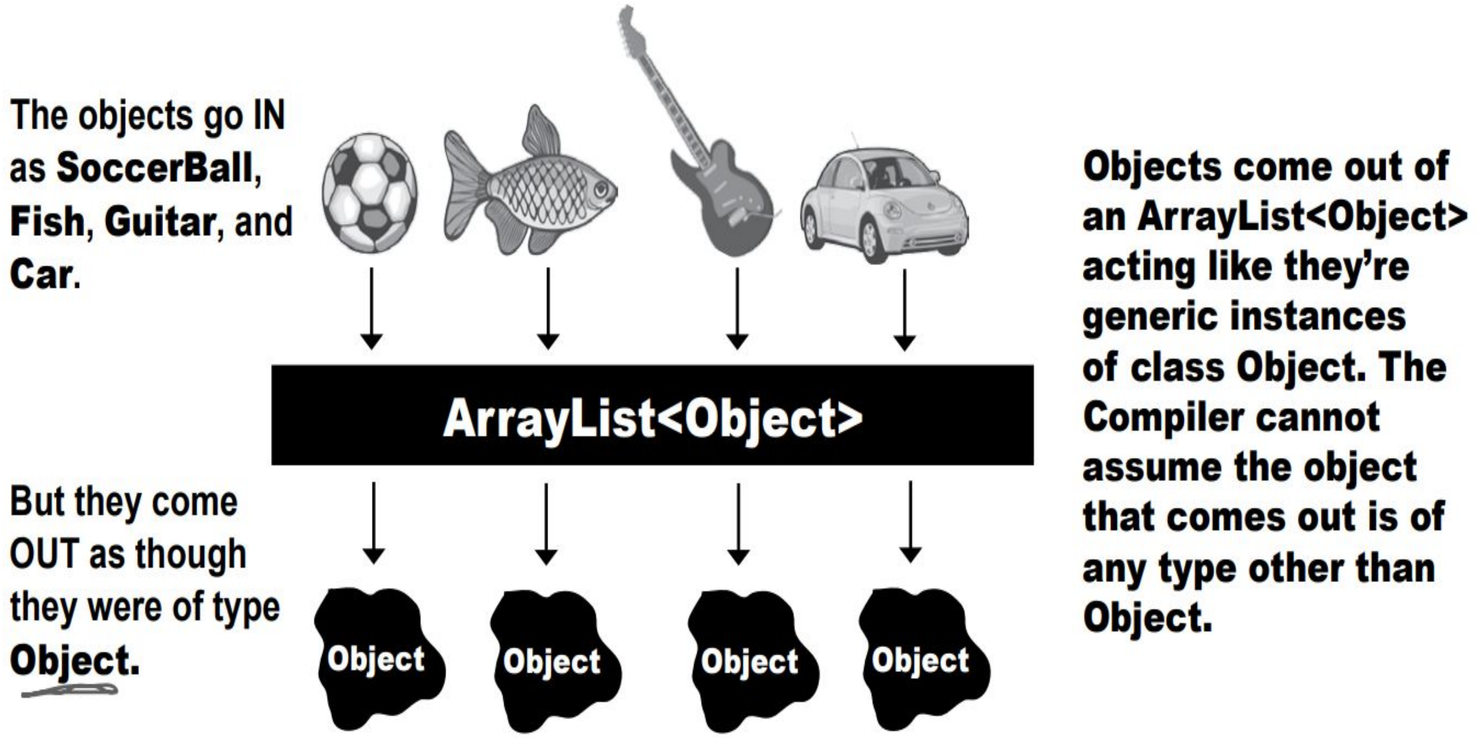
Make an ArrayList declared to hold Dog objects.

Make a Dog.

Add the Dog to the list.

Assign the Dog from the list to a new Dog reference variable. (Think of it as though the get() method declares a Dog return type because you used ArrayList<Dog>.)

Using polymorphic references of type Object

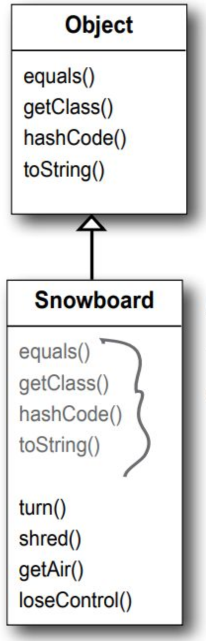


The objects go IN as **SoccerBall**, **Fish**, **Guitar**, and **Car**.

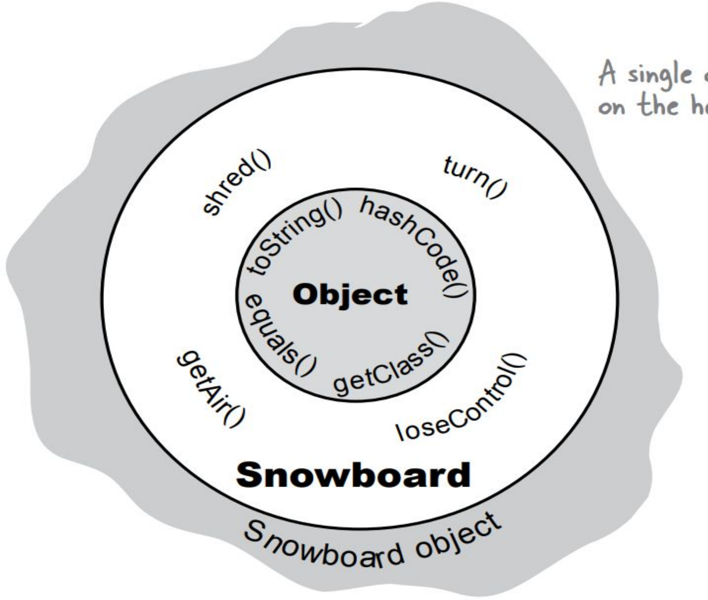
But they come OUT as though they were of type **Object**.

Objects come out of an ArrayList<Object> acting like they're generic instances of class Object. The Compiler cannot assume the object that comes out is of any type other than Object.

Using polymorphic references of type Object



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

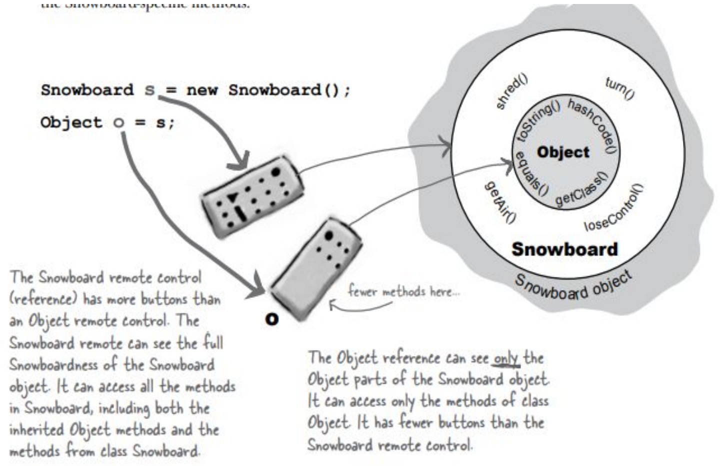


A single object on the heap.

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.

Using polymorphic references of type Object

1. ‘**Polymorphism**’ means ‘***many forms***’.
2. When you put an object in an ArrayList<Object>, you can treat it only as an Object, regardless of the type it was when you put it in.
3. When you get a reference from an ArrayList<Object>, the reference is always of type Object



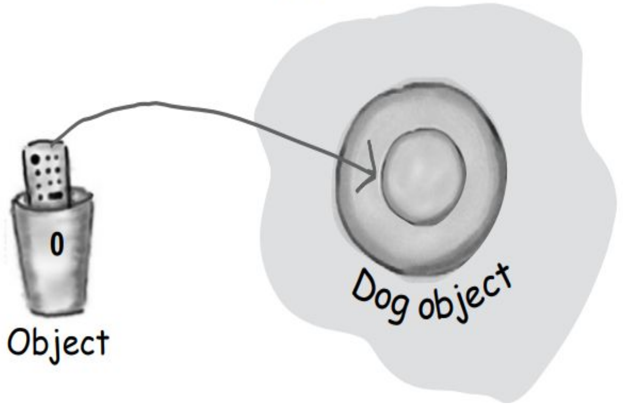
```
Snowboard s = new Snowboard();
Object o = s;
```

The Snowboard remote control (reference) has more buttons than an Object remote control. The Snowboard remote can see the full Snowboardness of the Snowboard object. It can access all the methods in Snowboard, including both the inherited Object methods and the methods from class Snowboard.

The Object reference can see only the Object parts of the Snowboard object. It can access only the methods of class Object. It has fewer buttons than the Snowboard remote control.

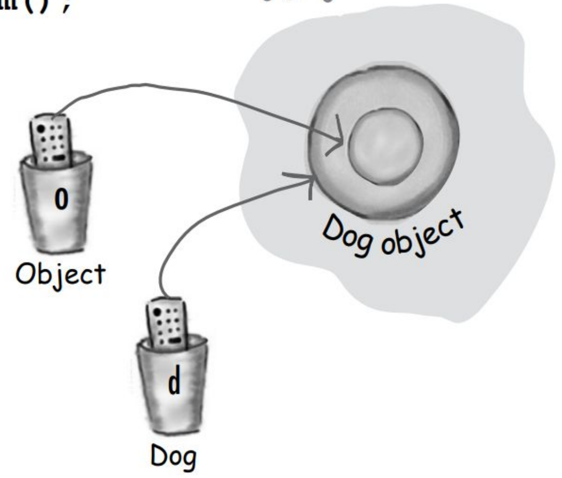
Casting Object

Casting an object reference back to its *real* type.

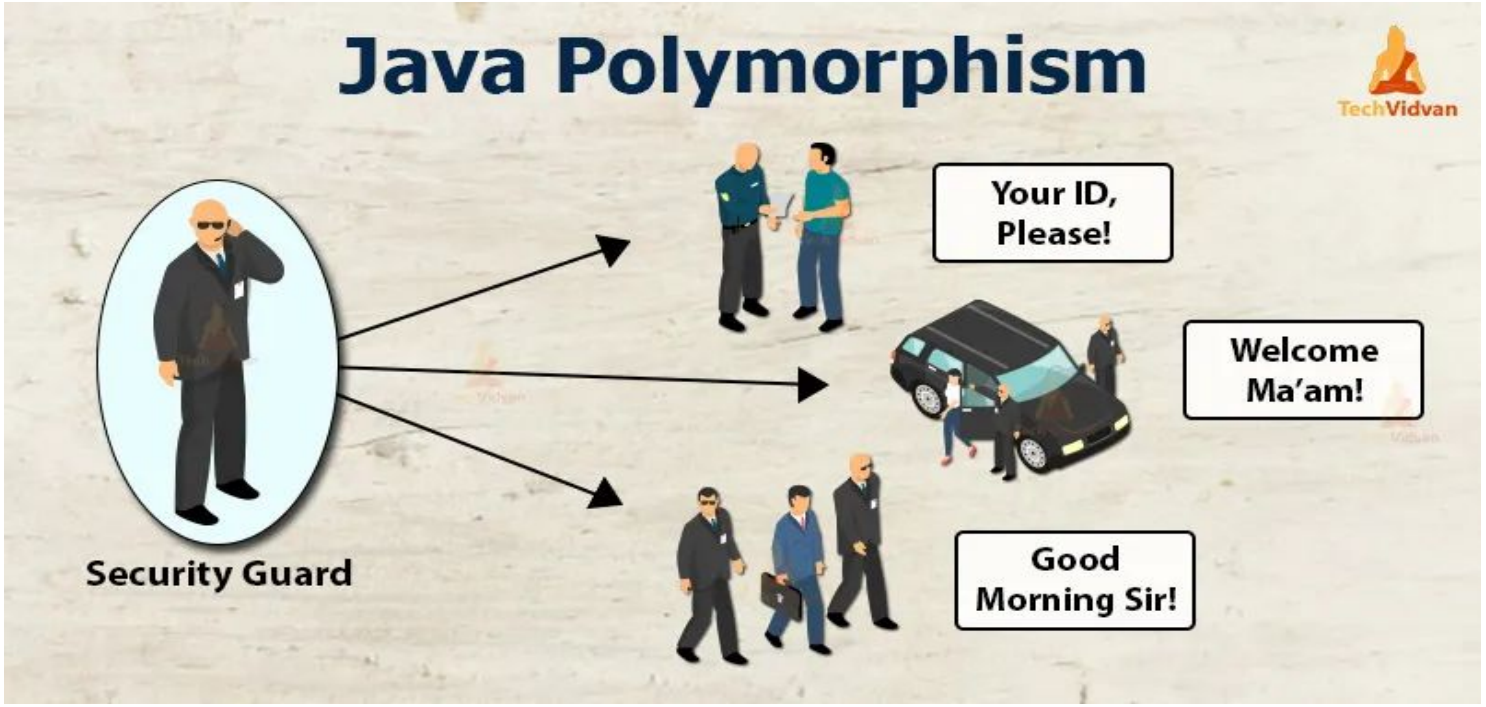


```
Object o = al.get(index);
Dog d = (Dog) o;
d.roam();
```

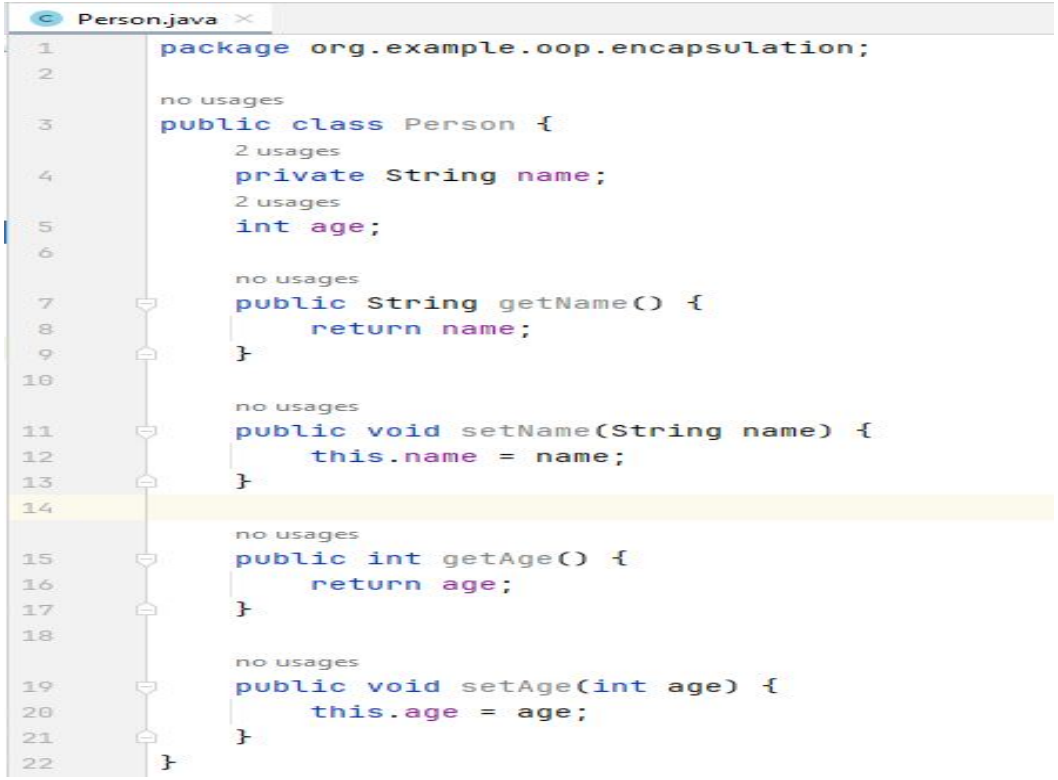
cast the Object back to a Dog we know is there.



‘Polymorphism’ means ‘*many forms*’.



Encapsulation


Person.java

```
package org.example.oop.encapsulation;

no usages
public class Person {
    2 usages
    private String name;
    2 usages
    int age;

    no usages
    public String getName() {
        return name;
    }

    no usages
    public void setName(String name) {
        this.name = name;
    }

    no usages
    public int getAge() {
        return age;
    }

    no usages
    public void setAge(int age) {
        this.age = age;
    }
}
```

Thank you!

Presented by

Temurmalik Nomozov

(temirmaliknomozov@gmail.com)