

# Inf1B

## Abstract Classes and Interfaces

Perdita Stevens

adapting earlier versions by Ewan Klein, Volker Seeker, et al.

School of Informatics

# Abstract Classes

# Greeting

- ▶ The function `makeGreeting` gets a greeting string from an object `greeter` of class `Hello`.
- ▶ Then sends a greeting to a friend.

## Talker

```
public static void makeGreeting>Hello greeter, String name) {  
    System.out.printf(greeter.sayHello() + ", " + name + "!");  
}  
public static void main(String[] args) {  
    Hello greeter = new Hello();  
    makeGreeting(greeter, "James");  
}
```

# Greeting

Hello class is trivial:

Hello

```
public class Hello {  
    public String sayHello() {  
        return "Hello";  
    }  
}
```

## Output

Hello, James!

## Extending Greeting

- ▶ Suppose we decide to go international, and add a new class `Bonjour`.
- ▶ Similar to `Hello`, but different method name and different return string.

### `Bonjour`

```
public class Bonjour {  
    public String ditBonjour() {  
        return "Bonjour";  
    }  
}
```

## Possible Solution?

- ▶ `Hello` and `Bonjour` should both be usable by `makeGreeting`
- ▶ But we can't do this straightforwardly; so create overload with a new 'French' version?

### Talker

```
public static void makeGreeting>Hello greeter, String name) {  
    System.out.printf(greeter.sayHello() + ", " + name + "!");  
}
```

```
public static void main(String[] args) {  
    Hello engGreeter = new Hello();  
    makeGreeting(engGreeter, "James");  
    Bonjour frGreeter = new Bonjour();  
    makeGreeting(frGreeter, "Jacques");  
}
```

## Possible Solution?

- ▶ `Hello` and `Bonjour` should both be usable by `makeGreeting`
- ▶ But we can't do this straightforwardly; so create overload with a new 'French' version?

### Talker

```
public static void makeGreeting(Hello greeter, String name) {
    System.out.printf(greeter.sayHello() + ", " + name + "!\n");
}
public static void makeGreeting(Bonjour greeter, String name) {
    System.out.printf(greeter.ditBonjour() + ", " + name + "!\n");
}
public static void main(String[] args) {
    Hello engGreeter = new Hello();
    makeGreeting(engGreeter, "James");
    Bonjour frGreeter = new Bonjour();
    makeGreeting(frGreeter, "Jacques");
}
```

# Greeting

- ▶ Overloading `makeGreeting` to use `Bonjour` is wasteful — we're duplicating code.
- ▶ Can we get a more general version of `makeGreeting` which can use both `Hello` and `Bonjour`?



# Greeting

- ▶ Overloading `makeGreeting` to use `Bonjour` is wasteful — we're duplicating code.
- ▶ Can we get a more general version of `makeGreeting` which can use both `Hello` and `Bonjour`?

**Step 1:** Give both these classes a common API; i.e., they should use the same methods.

# Greeting

Hello

```
public class Hello {  
    public String greet() {  
        return "Hello";  
    }  
}
```

Bonjour

```
public class Bonjour {  
    public String greet() {  
        return "Bonjour";  
    }  
}
```

## Greeting

- ▶ How do we say, in general, what the shared API is?
- ▶ For example, how to enforce that a new class `BuonGiorno` conforms to this API?

## Greeting

- ▶ How do we say, in general, what the shared API is?
- ▶ For example, how to enforce that a new class `BuonGiorno` conforms to this API?

**Step 2:** Pull the API into a superclass `Greeting`.

```
public class Hello extends Greeting {  
    public String greet() {  
        return "Hello";  
    }  
}
```

```
public class Bonjour extends Greeting {  
    public String greet() {  
        return "Bonjour";  
    }  
}
```

## Greeter

- ▶ How do we refactor `makeGreeting` to use objects that implement `Greeting`?

# Greeter

- ▶ How do we refactor `makeGreeting` to use objects that implement `Greeting`?

**Step 3:** Use `Greeting` as **polymorphic type** in the function signature.

## Talker

```
public static void makeGreeting(Greeting greeter, String name)
    System.out.printf(greeter.greet() + ", " + name + "!");
}
public static void main(String[] args) {
    Hello engGreeter = new Hello();
    makeGreeting(engGreeter, "James")
    Bonjour frGreeter = new Bonjour();
    makeGreeting(frGreeter, "Jacques");
}
```

## Greeting

But wait, something is not well defined. What happens in this case?

# Greeting

But wait, something is not well defined. What happens in this case?

## Talker

```
public static void makeGreeting(Greeting greeter, String name) {  
    System.out.printf(greeter.greet() + ", " + name + "!" );  
}  
public static void main(String[] args) {  
    Greeting greeter = new Greeting();  
    makeGreeting(greeter, "James")  
}
```



# Greeting

But wait, something is not well defined. What happens in this case?

## Talker

```
public static void makeGreeting(Greeting greeter, String name) {  
    System.out.printf(greeter.greet() + ", " + name + "!" );  
}  
public static void main(String[] args) {  
    Greeting greeter = new Greeting();  
    makeGreeting(greeter, "James")  
}
```

What does it print?

# Greeting

- ▶ Print output for general superclass `Greeting` is not sensible to have.

## Greeting

```
public class Greeting {  
    public String greet() {  
        return ???;  
    }  
}
```

# Greeting

- ▶ Print output for general superclass `Greeting` is not sensible to have.
- ▶ Therefore, we declare `Greeting` to be **abstract**

## Greeting

```
public abstract class Greeting {  
    public String greet() {  
        return ???;  
    }  
}
```

# Greeting

- ▶ Print output for general superclass `Greeting` is not sensible to have.
- ▶ Therefore, we declare `Greeting` to be **abstract**
- ▶ and provide no superclass implementation for `greet`.

## Greeting

```
public abstract class Greeting {  
    public abstract String greet();  
}
```

# Greeting

- ▶ Print output for general superclass `Greeting` is not sensible to have.
- ▶ Therefore, we declare `Greeting` to be **abstract**
- ▶ and provide no superclass implementation for `greet`.

## Greeting

```
public abstract class Greeting {  
    public abstract String greet();  
}
```

This solves our class design problem.

## Greeting

- ▶ Instantiation of an **abstract** class is not allowed.

# Greeting

- ▶ Instantiation of an **abstract** class is not allowed.

## Talker

```
public static void makeGreeting(Greeting greeter, String name) {  
    System.out.printf(greeter.greet() + ", " + name + "!!");  
}  
public static void main(String[] args) {  
    Greeting greeter = new Greeting();  
    makeGreeting(greeter, "James")  
}
```

This causes a compiler error:

**error: Greeting is abstract; cannot be instantiated**

# Greeting

- ▶ Instantiation of an **abstract** class is not allowed.
- ▶ The **abstract** method `greet` enforces required API for each subclass.



# Greeting

- ▶ Instantiation of an **abstract** class is not allowed.
- ▶ The **abstract** method `greet` enforces required API for each subclass.

```
public class Hello extends Greeting {  
    // must override abstract method  
    // to avoid compiler error  
    public String greet() {  
        return "Hello";  
    }  
}
```

# Animal Objects?

## Creating new objects

```
Wolf wolfie = new Wolf();
```

```
Animal leo = new Lion();
```

```
Animal weird = new Animal();
```

- ▶ Animal class is meant to contain information that all animals have in common.
- ▶ But this is not enough to define any one specific animal.

# Concrete vs. Abstract

## Concrete

- ▶ Examples: Cat, Wolf, Hello
- ▶ Specific enough to be instantiated.

## Abstract

- ▶ Examples: Animal, Feline, Greeting
- ▶ Not intended to have instances.
- ▶ Only useful if extended.
- ▶ Any 'instances' will have to be instances of a **subclass** of the abstract class.

# The Abstract Animal

## Animal

```
public abstract class Animal {  
    public void sleep() {  
        System.out.println("Sleeping: Zzzzz");  
    }  
    public void makeNoise() {  
        System.out.println("Noises...");  
    }  
    public void roam() {  
        System.out.println("Roamin' on the plain.");  
    }  
}
```

Just put the keyword `abstract` before the class declaration.

## The Abstract Animal

- ▶ An abstract class can be extended by other abstract classes.
- ▶ Canine and Feline can (and should) both be abstract.

### Animal

```
public abstract class Animal {  
    public void sleep() {  
        System.out.println("Sleeping: Zzzzz");  
    }  
    public void makeNoise() {  
        System.out.println("Noises...");  
    }  
    public void roam() {  
        System.out.println("Roamin' on the plain.");  
    }  
}
```

Just put the keyword `abstract` before the class declaration.

# The Abstract Animal

## Animal

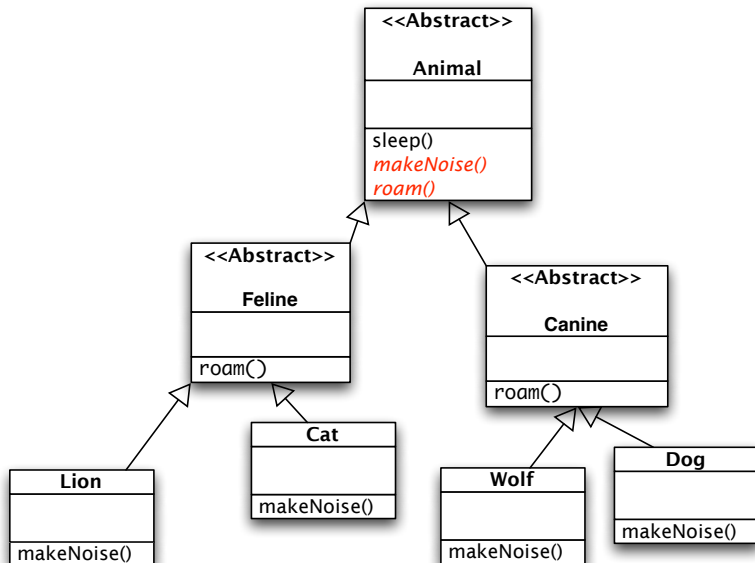
```
public abstract class Animal {  
    public void sleep() {  
        System.out.println("Sleeping: Zzzzz");  
    }  
    public abstract void roam();  
    public abstract void makeNoise();  
}
```

Now has abstract methods!

# The Abstract Animal

- ▶ `roam()` and `makeNoise()` are abstract methods:
  - ▶ no body;
  - ▶ **must** be implemented in any concrete subclass (implemented ~ overridden);
  - ▶ don't have to be implemented by an abstract subclass;
  - ▶ can only be declared in an abstract class;
- ▶ `sleep()` is not abstract, so can be straightforwardly inherited.

# Abstract Classes in Animal Hierarchy





## Using Abstract Classes

- ▶ Use an abstract class when you have several similar classes that:
  - ▶ have a lot in common — the implemented parts of the abstract class
  - ▶ have some differences — the abstract methods.

Let's practise that



<https://www.theodysseyonline.com/your-brain-is-muscle-exercise-it>

# What does it print?

```
public abstract class Vehicle {
    public void drive() {
        System.out.println("drivedrive");
    }
}

public class Car extends Vehicle {
    public void drive() {
        System.out.println("rollroll");
    }
}

public class Main {
    public static void main(String[] args) {
        Vehicle c = new Vehicle();
        c.drive();
    }
}
```

## What does it print?

```
public abstract class Vehicle {
    public void drive() {
        System.out.println("drivedrive");
    }
}

public class Car extends Vehicle {
    public void drive() {
        System.out.println("rollroll");
    }
}

public class Main {
    public static void main(String[] args) {
        Vehicle c = new Vehicle();
        c.drive();
    }
}
```

Does not compile because abstract classes cannot be instantiated.

## What does it print?

```
public abstract class Vehicle {
    public abstract void drive();
}

public class Car extends Vehicle {
}

public class Main {
    public static void main(String[]
        args) {
        Car c = new Car();
        c.drive();
    }
}
```

## What does it print?

```
public abstract class Vehicle {
    public abstract void drive();
}

public class Car extends Vehicle {
}

public class Main {
    public static void main(String[]
        args) {
        Car c = new Car();
        c.drive();
    }
}
```

Does not compile because abstract methods need to be implemented in subclass.

## What does it print?

```
public abstract class Vehicle {
    public abstract void drive();
}

public abstract class Car extends Vehicle {
    public void drive() {
        System.out.println("rollroll");
    }
}

public class Polo extends Car {
}

public class Main {
    public static void main(String[] args) {
        Polo p = new Polo();
        p.drive();
    }
}
```

## What does it print?

```
public abstract class Vehicle {
    public abstract void drive();
}

public abstract class Car extends Vehicle {
    public void drive() {
        System.out.println("rollroll");
    }
}

public class Polo extends Car {
}

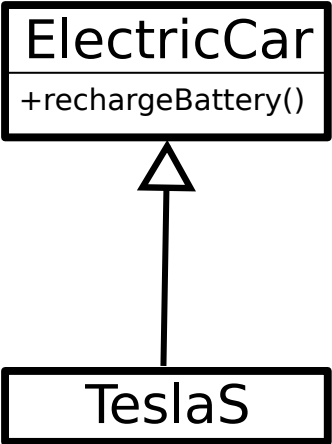
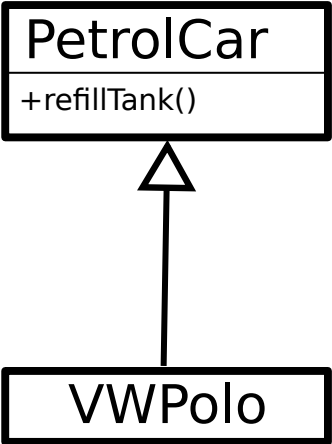
public class Main {
    public static void main(String[] args) {
        Polo p = new Polo();
        p.drive();
    }
}
```

Prints **rollroll**. Car implements the abstract method and Polo inherits this implementation.

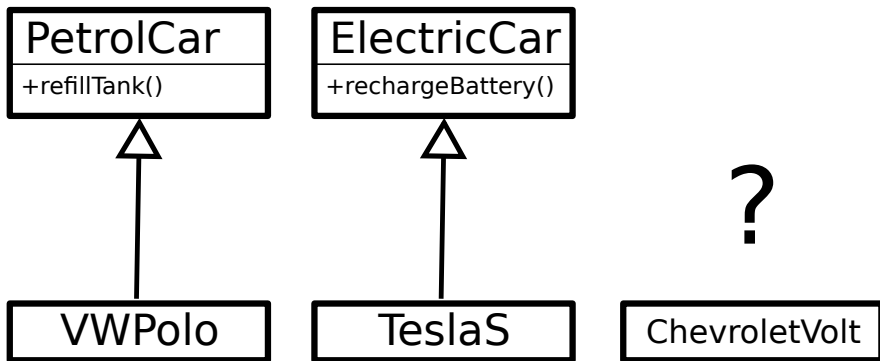


# Interfaces

# Different Types of Cars

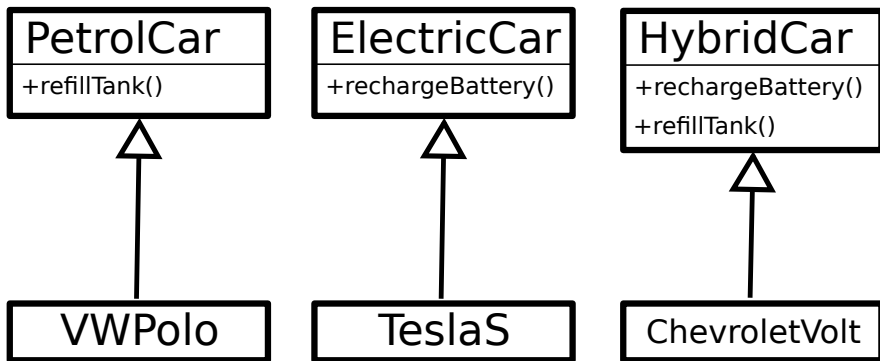


## Hybrid Car



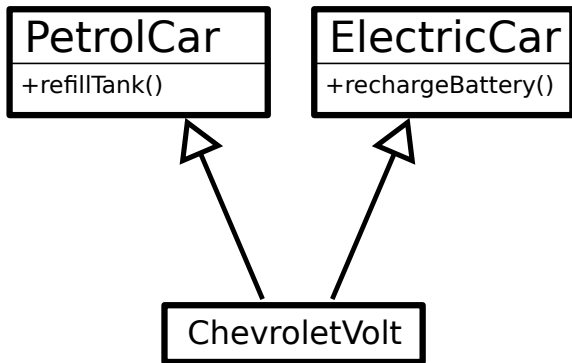
How to handle a plug-in Hybrid which has both battery and petrol,  
i.e. **features of both superclasses?**

## Hybrid Car



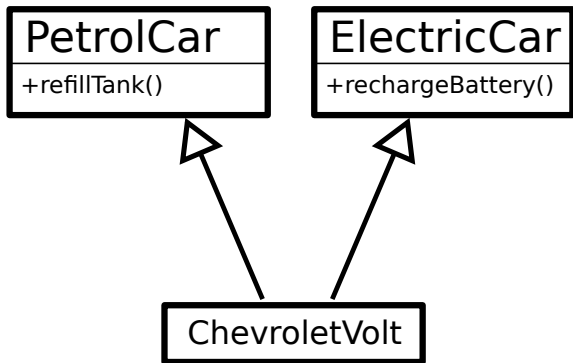
Creating a new superclass with both methods would be wasteful -  
**code duplication.**

## Multiple Inheritance



Inheriting from **both** classes would be best.

## Multiple Inheritance

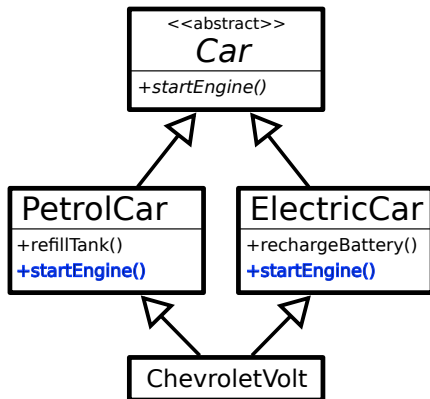


Inheriting from **both** classes would be best.

Unfortunately, multiple inheritance has some ambiguity problems.

# Ambiguity Problems with Multiple Inheritance

## The Deadly Diamond of Death



- ▶ both `PetrolCar` and `ElectricCar` override `startEngine`
- ▶ which version of `startEngine` does `ChevroletVolt` inherit?

## Multiple Inheritance Support

Some languages **resolve ambiguity** using a complex implementation, e.g. C++



## Multiple Inheritance Support

Some languages **resolve ambiguity** using a complex implementation, e.g. C++

Java, **avoids ambiguity** by using **Interfaces**

# Interfaces in Java

- ▶ Interfaces are defined using the `interface` keyword
- ▶ like abstract classes they cannot be instantiated
- ▶ unlike abstract classes **all methods** have to be abstract

```
public interface PetrolCar {  
    public abstract void refillTank();  
}
```

```
public interface ElectricCar {  
    public abstract void rechargeBattery();  
}
```

# Interfaces in Java

- ▶ Interfaces are defined using the `interface` keyword
- ▶ like abstract classes they cannot be instantiated
- ▶ unlike abstract classes **all methods** have to be abstract

```
public interface PetrolCar {  
    public abstract void refillTank();  
}
```

```
public interface ElectricCar {  
    public abstract void rechargeBattery();  
}
```

They do not allow sharing of implementations but enforce an API.

# Interfaces in Java

- ▶ classes can **implement** interfaces by using the **implements** keyword
- ▶ an implementation for each method is enforced by the compiler

```
public class ChevroletVolt implements PetrolCar, ElectricCar {  
    public void refillTank() {  
        // refill petrol  
    }  
  
    public void rechargeBattery() {  
        // recharge power  
    }  
}
```

# Interfaces in Java

- ▶ classes can **implement** interfaces by using the **implements** keyword
- ▶ an implementation for each method is enforced by the compiler

```
public class ChevroletVolt implements PetrolCar, ElectricCar {  
    public void refillTank() {  
        // refill petrol  
    }  
  
    public void rechargeBattery() {  
        // recharge power  
    }  
}
```

Both extension and implementation is possible:

```
public class ChevroletVolt extends Chevrolet implements PetrolCar, ElectricCar
```

# Syntactic Sugar

- ▶ all methods in an interface must be **public**

```
public interface PetrolCar {  
    public abstract void refillTank();  
}
```

# Syntactic Sugar

- ▶ all methods in an interface must be **public**

```
public interface PetrolCar {  
    abstract void refillTank();  
}
```

# Syntactic Sugar

- ▶ all methods in an interface must be **public**
- ▶ all methods in an interface must be **abstract**

```
public interface PetrolCar {  
  
    abstract void refillTank();  
}
```



# Syntactic Sugar

- ▶ all methods in an interface must be **public**
- ▶ all methods in an interface must be **abstract**

```
public interface PetrolCar {  
  
    void refillTank();  
}
```

# Syntactic Sugar

- ▶ all methods in an interface must be **public**
- ▶ all methods in an interface must be **abstract**
- ▶ no constructors are allowed

```
public interface PetrolCar {  
  
    void refillTank();  
}
```

# Syntactic Sugar

- ▶ all methods in an interface must be **public**
- ▶ all methods in an interface must be **abstract**
- ▶ no constructors are allowed
- ▶ members are allowed but they must be **public static final**

```
public interface PetrolCar {  
    public static final String FUEL = "Diesel";  
    void refillTank();  
}
```

# Syntactic Sugar

- ▶ all methods in an interface must be **public**
- ▶ all methods in an interface must be **abstract**
- ▶ no constructors are allowed
- ▶ members are allowed but they must be **public static final**

```
public interface PetrolCar {  
    String FUEL = "Diesel";  
    void refillTank();  
}
```

## Avoiding Code Duplication

When using interfaces for `PetrolCar` and `ElectricCar` we would have to implement `refillTank` and `rechargeBattery` for each new superclass.

## Avoiding Code Duplication

When using interfaces for `PetrolCar` and `ElectricCar` we would have to implement `refillTank` and `rechargeBattery` for each new superclass.

To avoid this in Java, you could use object **Composition**.

## Avoiding Code Duplication

When using interfaces for `PetrolCar` and `ElectricCar` we would have to implement `refillTank` and `rechargeBattery` for each new superclass.

To avoid this in Java, you could use object **Composition**.

```
public class ChevroletVolt implements PetrolCar, ElectricCar {  
    private final BatteryCharger charger;  
    private final FuelPump pump;  
  
    public void refillTank() {  
        pump.refill();  
    }  
  
    public void rechargeBattery() {  
        charger.charge();  
    }  
}
```

## Inheritance in Java API

Inheritance using interfaces and abstract classes is used a lot in the Java API.

Have a browse:

`https:`

`//docs.oracle.com/en/java/javase/15/docs/api/`



## Comparable Interface

You have an `ArrayList` of cows and you want to order them by size.

```
public class Cow extends Animal {  
    private int size;  
    private float milkYield;  
    private String name;  
    ...  
}
```

## Comparable Interface

Java provides a convenient method `Collections.sort()` in `java.util.Collections`.

```
ArrayList<Cow> herd = collectCows();  
Collections.sort(herd); // sorts the herd
```

## Comparable Interface

Java provides a convenient method `Collections.sort()` in [java.util.Collections](#).

```
ArrayList<Cow> herd = collectCows();  
Collections.sort(herd); // sorts the herd
```

How does the sort method know that you want to order by size and not by milkYield or name?

## Comparable Interface

The `sort` method expects objects to implement the **`java.lang.Comparable`** interface.

The `Comparable` interface forces subclasses to implement the **`compareTo`** method.

# Comparable Interface

The sort method expects objects to implement the **java.lang.Comparable** interface.

The **Comparable** interface forces subclasses to implement the **compareTo** method.

```
public class Cow extends Animal implements Comparable<Cow>{
    private int size;
    private float milkYield;
    private String name;

    @Override
    public int compareTo(Cow other) {
        ...
    }
    ...
}
```

## compareTo Method

compareTo is expected to be used in the following way:

- ▶ if this is less than other, return a negative number
- ▶ if this is greater than other, return a positive number
- ▶ if this is equal to other, return zero

## compareTo Method

compareTo is expected to be used in the following way:

- ▶ if this is less than other, return a negative number
- ▶ if this is greater than other, return a positive number
- ▶ if this is equal to other, return zero

```
public class Cow extends Animal implements Comparable<Cow>{
    private int size;
    private float milkYield;
    private String name;

    @Override
    public int compareTo(Cow other) {
        if (size < other.size) return -1;
        else if (size > other.size) return 1;
        else return 0;
    }
    ...
}
```

## compareTo Method

compareTo is expected to be used in the following way:

- ▶ if this is less than other, return a negative number
- ▶ if this is greater than other, return a positive number
- ▶ if this is equal to other, return zero

```
public class Cow extends Animal implements Comparable<Cow>{
    private int size;
    private float milkYield;
    private String name;

    @Override
    public int compareTo(Cow other) {
        return size - other.size;
    }
    ...
}
```

This works but is bad style!



## compareTo Method

compareTo is expected to be used in the following way:

- ▶ if this is less than other, return a negative number
- ▶ if this is greater than other, return a positive number
- ▶ if this is equal to other, return zero

```
public class Cow extends Animal implements Comparable<Cow>{
    private int size;
    private float milkYield;
    private String name;

    @Override
    public int compareTo(Cow other) {
        return size - other.size;
    }
    ...
}
```

This works but is bad style!

Integer overflow possible

## CompareTo Method

CompareTo is expected to be used in the following way:

- ▶ if this is less than other, return a negative number
- ▶ if this is greater than other, return a positive number
- ▶ if this is equal to other, return zero

```
public class Cow extends Animal implements Comparable<Cow>{
    private int size;
    private float milkYield;
    private String name;

    @Override
    public int compareTo(Cow other) {
        return Integer.compare(size, other.size);
    }
    ...
}
```

Java's boxed primitives have static compare methods.

## CompareTo Method

CompareTo is expected to be used in the following way:

- ▶ if this is less than other, return a negative number
- ▶ if this is greater than other, return a positive number
- ▶ if this is equal to other, return zero

```
public class Cow extends Animal implements Comparable<Cow>{
    private int size;
    private float milkYield;
    private String name;

    @Override
    public int compareTo(Cow other) {
        return name.compareTo(other.name);
    }
    ...
}
```

Many API classes such as boxed primitives or [String](#) implement the [Comparable](#) interface already.

# Let's practise that



<https://www.theodysseyonline.com/your-brain-is-muscle-exercise-it>

## What does it print?

```
public interface Drivable {
    public void startEngine() {
        System.out.println("WrummWrumm");
    }
    public abstract void drive();
}

public class Car implements Drivable {
    public void drive() {
        super.startEngine();
        System.out.println("rollroll");
    }
}

public class Main {
    public static void main(String[] args) {
        Car c = new Car();
        c.drive();
    }
}
```

## What does it print?

```
public interface Drivable {
    public void startEngine() {
        System.out.println("WrummWrumm");
    }
    public abstract void drive();
}

public class Car implements Drivable {
    public void drive() {
        super.startEngine();
        System.out.println("rollroll");
    }
}

public class Main {
    public static void main(String[] args) {
        Car c = new Car();
        c.drive();
    }
}
```

Does not compile  
because interfaces are  
not allowed to  
implement methods.

## What does it print?

```
public interface Drivable {
    public abstract void drive();
}

public class Car implements Drivable {
    public void drive() {
        System.out.println("rollroll");
    }
}

public class Polo extends Car {
    public void drive() {
        System.out.println("PoloPower");
    }
}

public class Main {
    public static void main(String[] args) {
        Polo p = new Polo();
        p.drive();
    }
}
```

## What does it print?

```
public interface Drivable {
    public abstract void drive();
}

public class Car implements Drivable {
    public void drive() {
        System.out.println("rollroll");
    }
}

public class Polo extends Car {
    public void drive() {
        System.out.println("PoloPower");
    }
}

public class Main {
    public static void main(String[] args) {
        Polo p = new Polo();
        p.drive();
    }
}
```

Prints **PoloPower**. Car implements the drive method and Polo overrides it.



# Summary

- ▶ abstract classes can be used to implement common behaviour without allowing instantiation (concrete vs. abstract)
- ▶ abstract methods can be used to enforce API on subclasses
- ▶ interfaces allow multiple inheritance but cannot be used to implement behaviour

# Reading

## Objects First

Chapter 12 *Further Abstraction Techniques*