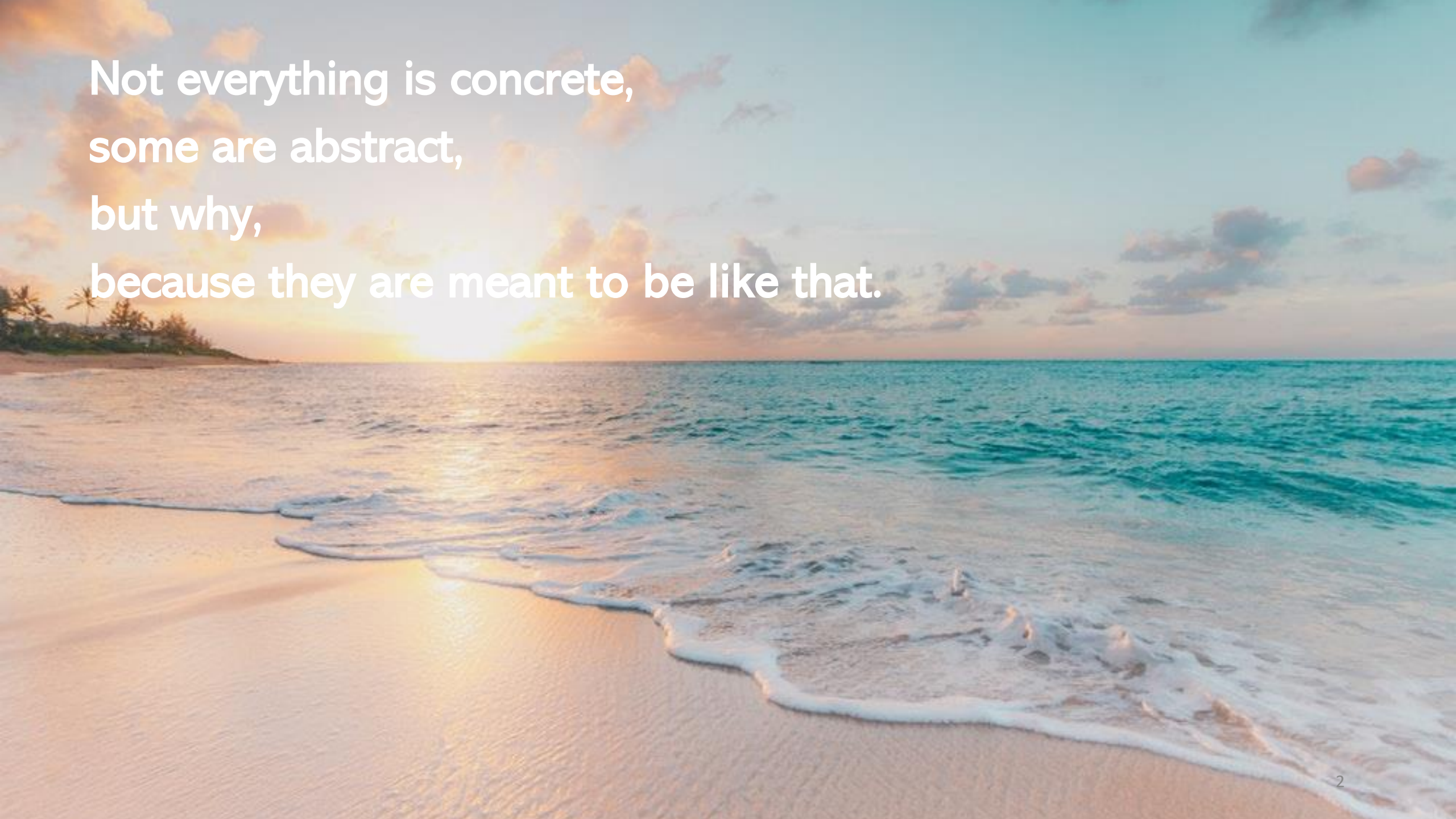
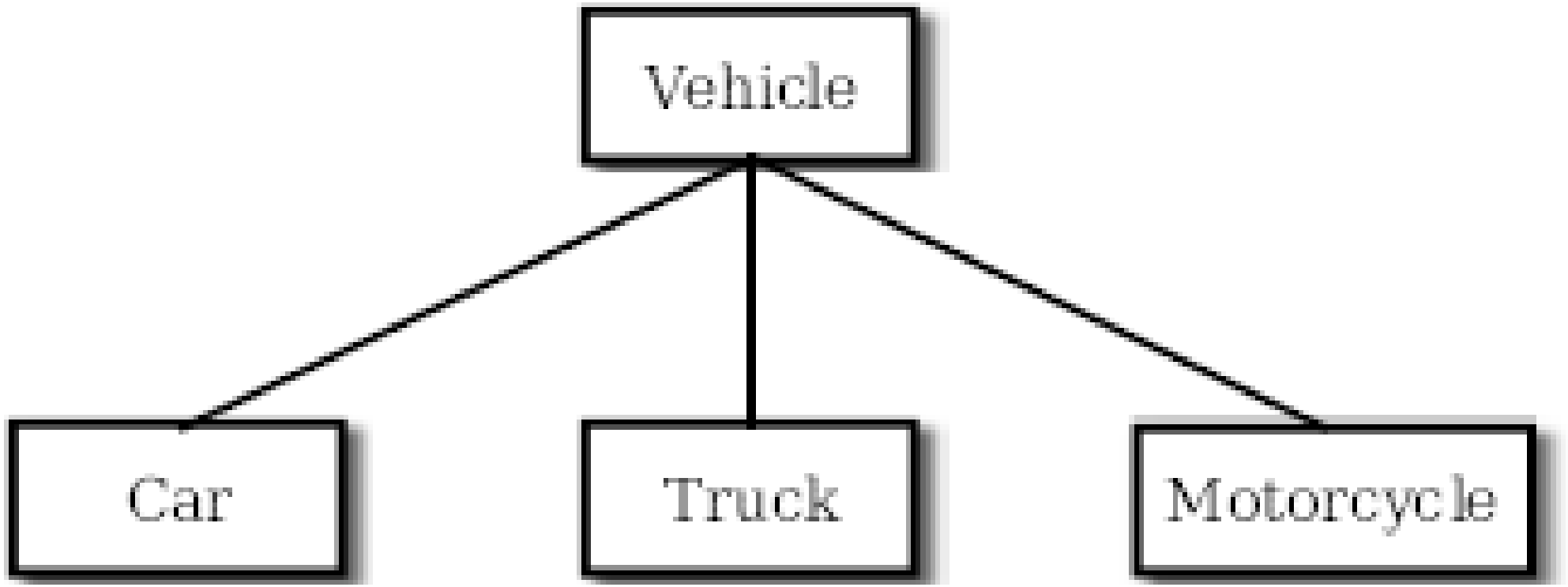
# Foundations of Programming 2: Abstract Classes and Interfaces

**FoP 2 Teaching Team, Faculty of Computer Science, Universitas Indonesia**
**Correspondence: Fariz Darari (fariz@cs.ui.ac.id)**

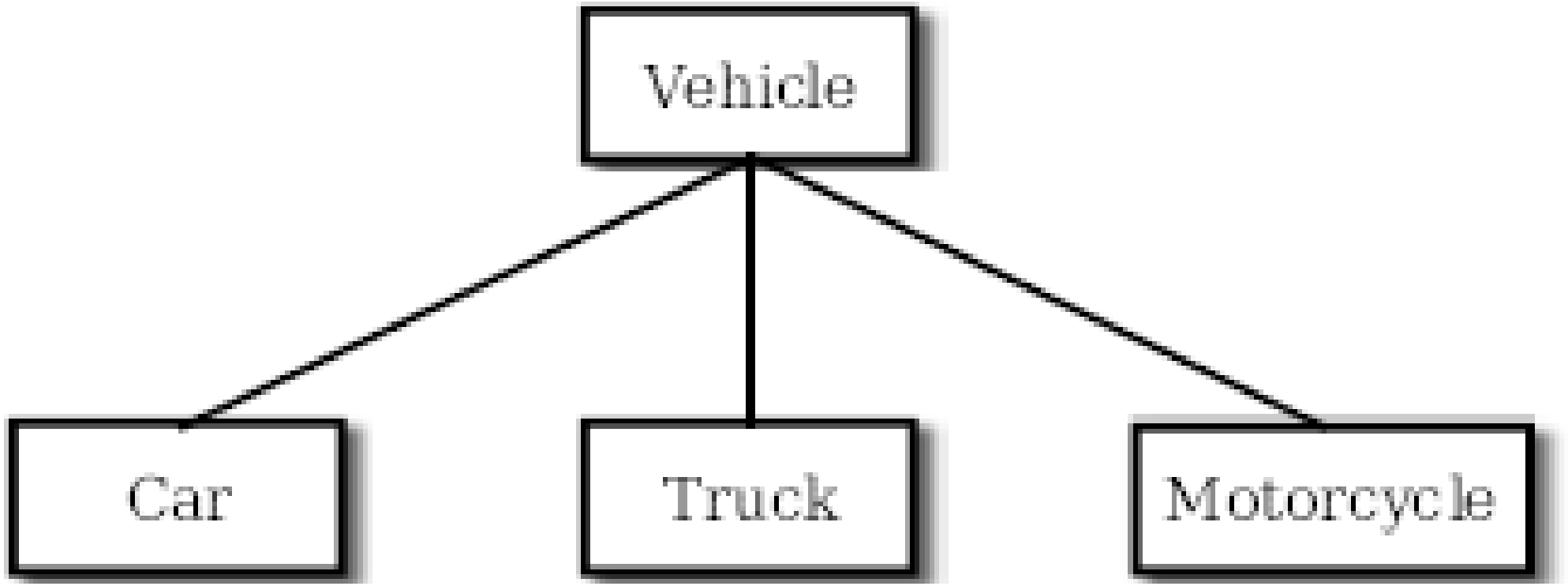
UNIVERSITAS INDONESIA
FAKULTAS ILMU KOMPUTER


Not everything is concrete,
some are abstract,
but why,
because they are meant to be like that.

# Why?



# Why?




Does it really make sense to make an object of the class Vehicle?
I mean, how could you have a vehicle without knowing if it's a car,
a truck, or a motorcycle?

# Abstract class: Rules

- An abstract class is abstract, in the sense that you cannot concretize it (that is, you cannot instantiate any object).
- However (and this is the reason abstract classes can be useful) , you can create a class extending an abstract class.
- An abstract class may have abstract methods (beside the standard ones), while an abstract method must be in an abstract class.
- A class extending an abstract class must implement all the abstract methods.

# Abstract method

An abstract method is abstract, in the sense that it is not implemented, like this:

```
abstract void hello(String name);
```

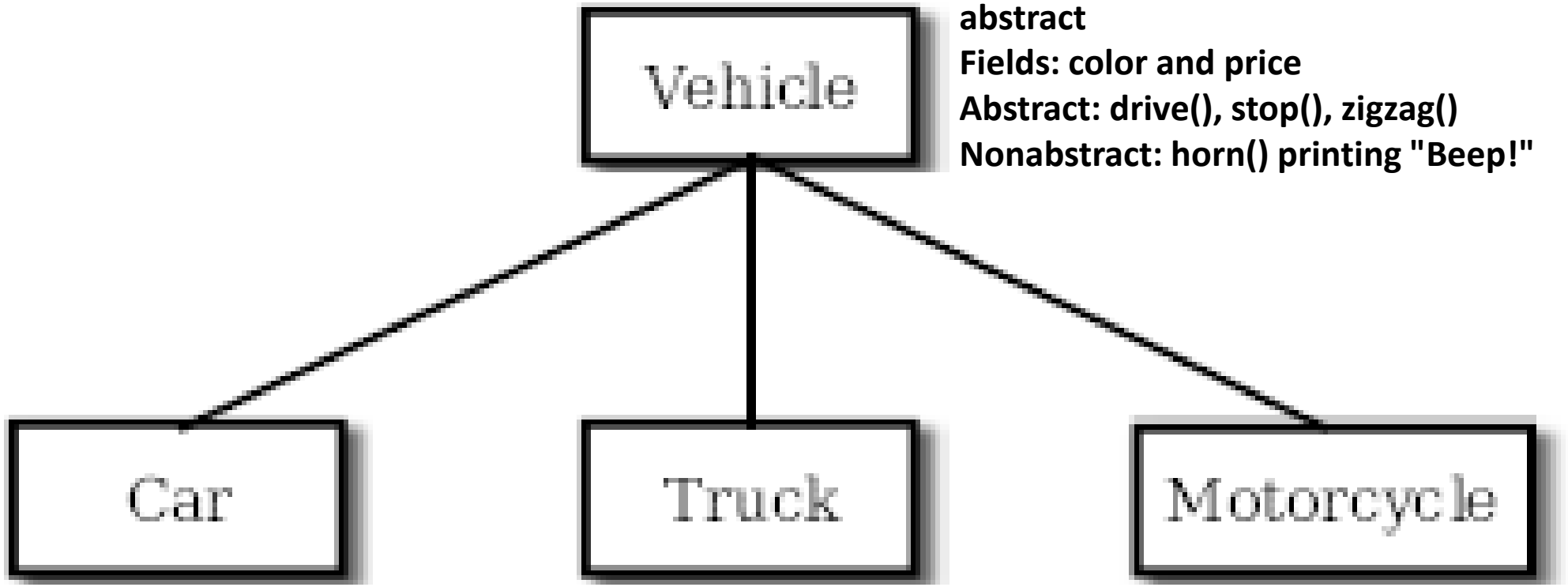
# Quiz time: Guess the output (not here, later)

```
public abstract class SimpleAbstract {
	int x;
	// an abstract class can have a constructor
	SimpleAbstract() {
		System.out.println("Abstract");
	}
	abstract void hello(String name); // abstract method
	// an abstract class can have a standard method
	void concrete() {
		System.out.println("Concrete in abstract");
	}
}
```

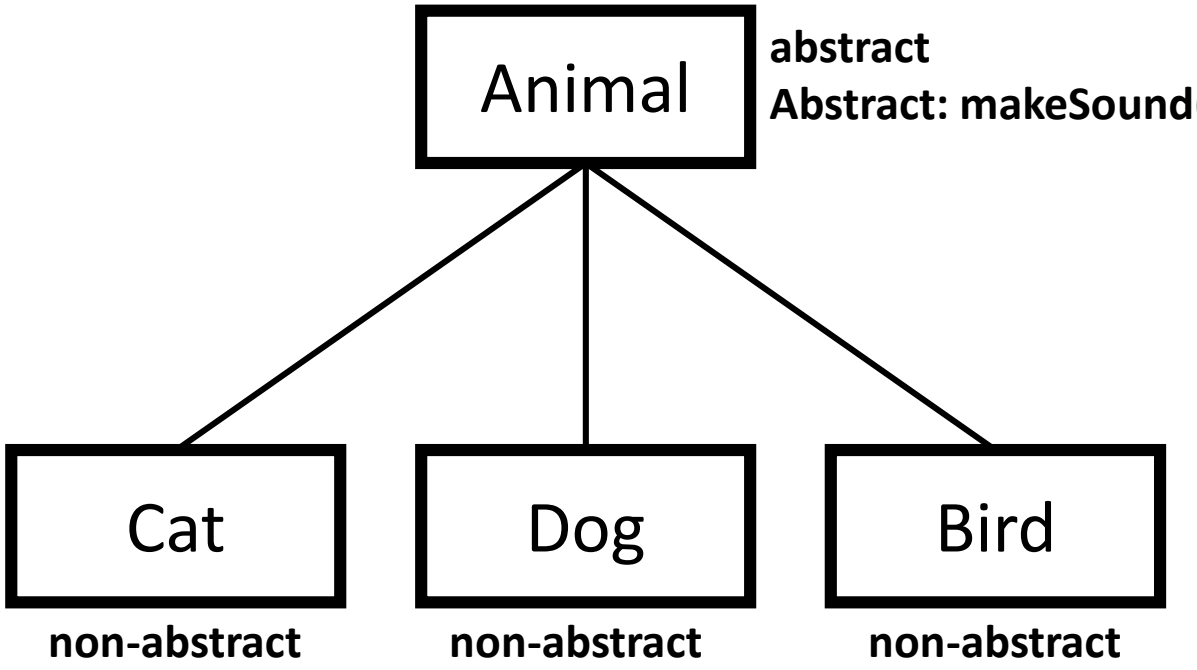
# Quiz time: Now guess the output

```
public class Simple extends SimpleAbstract {
	// try to comment this
	void hello(String name) {
		System.out.println("Hello, " + name);
	}
	public static void main(String[] args) {
		// try to uncomment this
//		SimpleAbstract sa = new SimpleAbstract();
		// guess output
		Simple s = new Simple();
		s.hello("Sis");
		System.out.println(s.x);
		s.concrete();
	}
}
```

# Quiz time: Now try to implement this.



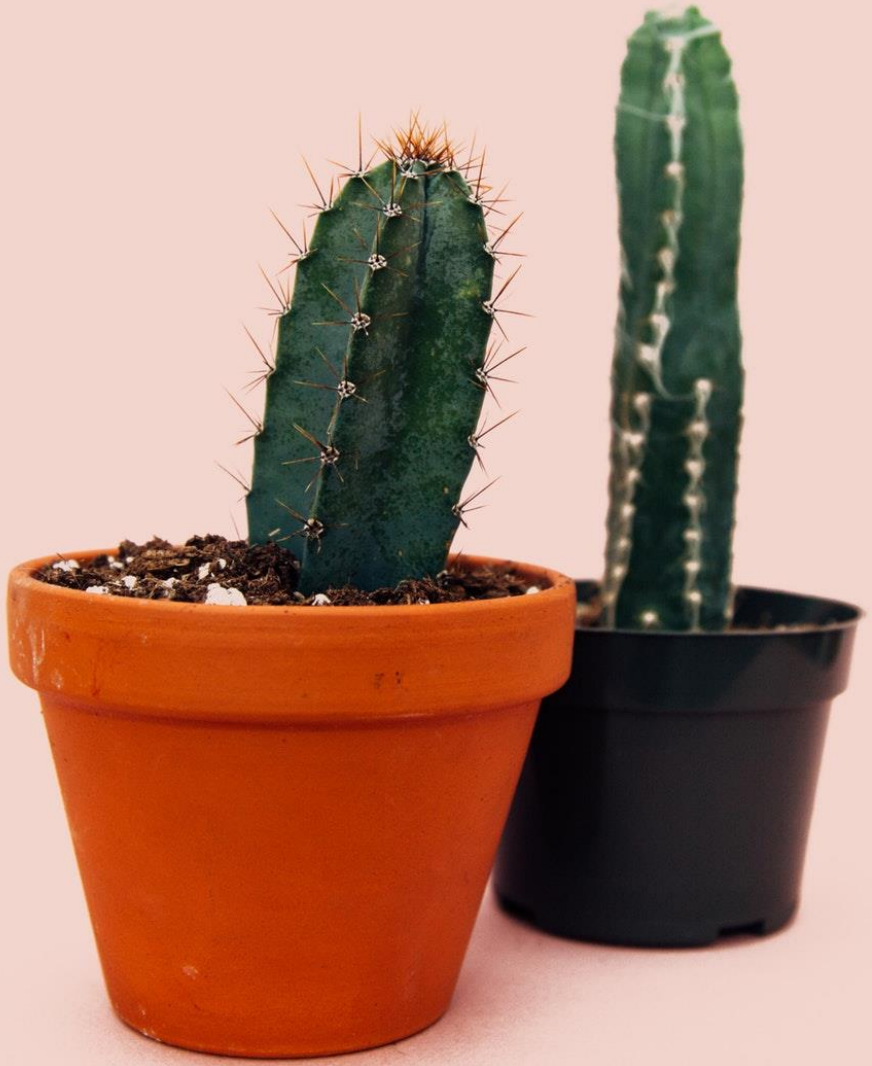
# Quiz time: Now try to implement this.



# Quiz time: Difference between nonabstract and abstract classes?

# Quiz time: Difference between nonabstract and abstract classes?

- Nonabstract classes can be instantiated, while abstract cannot.
- Nonabstract classes cannot enforce subclasses to implement (abstract) methods, while abstract can.



# Interfaces

# Interfaces

- Interfaces define a contract:
  What a class can do, but (generally) not the how
- As opposed to (abstract) classes, a class may "extends"
  multiple interfaces. Well, **it is now called "implements",** not "extends".
- Methods in interfaces are implicitly abstract and public.
- Interfaces **cannot** have constructors.
- Interfaces are like features, for example, you can add features
  to your smartphone, like GPS-featured and radio-featured.
- Similarly, Rentable could be an interface for Car, and Book; and
  Edible could be an interface for Mushroom, and Chicken.
- Observe that the classes implementing an inteface could be totally different.

# Interfaces for Cellphone

```
public interface GPSEnabled { // put in separate source code
    public void printLocation();
}

public interface RadioEnabled { // put in separate source code
    public void startRadio();
    public void stopRadio();
}

// put in separate source code
public class Cellphone implements GPSEnabled, RadioEnabled {
    public void printLocation() {
      System.out.println("Location");
    }
    public void startRadio() {
      System.out.println("Radio is ON!");
    }
    public void stopRadio() {
      System.out.println("Radio is OFF!");
    } }
```

# Quiz time: What is the output?

```
// .. put inside Cellphone.java
public static void main(String[] args) {

    Cellphone ciaoMi = new Cellphone();
    ciaoMi.printLocation();
    ciaoMi.startRadio();
    GPSEnabled samsu = new Cellphone();
    samsu.printLocation();
    samsu.startRadio();

}
```

# Quiz time

- Create an interface of AIEnabled
- The interface should include the following methods:
  - turnonChatbot(): turn on chatbot module
  - turnoffChatbot(): turn off chatbot module
  - checkChatbot(): check if chatbot module is on
- Implement AIEnabled in Cellphone
- Test your code

# Solution: AIEnabled.java

```
public interface AIEnabled {

    // turn on chatbot module
    void turnonChatbot();

    // turn off chatbot module
    void turnoffChatbot();

    // check if chatbot module is on
    boolean checkChatbot();

}
```

# Solution: Cellphone.java

```
public class Cellphone implements AIEnabled {

// ...

  boolean chatbotModule = false;
  public void turnonChatbot() { chatbotModule = true; }

  public void turnoffChatbot() { chatbotModule = false; }

  public boolean checkChatbot() { return chatbotModule; }

}
```

# Quiz time: List, ArrayList, and LinkedList

```
public static void main(String[] args) {

      List<String> lst1 = new ArrayList<String>();
      lst1.add("A");lst1.add("B");lst1.add("C");
      System.out.println(lst1.contains("A"));
      ((ArrayList<String>) lst1).ensureCapacity(100);
      System.out.println(lst1);

      List<String> lst2 = new LinkedList<String>();
      lst2.add("A");lst2.add("B");lst2.add("C");
      System.out.println(lst1.contains("A"));
      ((LinkedList<String>) lst2).addFirst("9");
      System.out.println(lst2);
}
```

*What can you observe?*

Both AL and LL are lists, but implemented differently:
https://dzone.com/storage/temp/895349-arraylist-linkedlistt.png

# Quiz time: List, ArrayList, and LinkedList

```
public static void main(String[] args) {

      List<String> lst1 = new ArrayList<String>();
      lst1.add("A");lst1.add("B");lst1.add("C");
      System.out.println(lst1.contains("A"));
      ((ArrayList<String>) lst1).ensureCapacity(100);
      System.out.println(lst1);

      List<String> lst2 = new LinkedList<String>();
      lst2.add("A");lst2.add("B");lst2.add("C");
      System.out.println(lst1.contains("A"));
      ((LinkedList<String>) lst2).addFirst("9");
      System.out.println(lst2);
}
```

Both AL and LL are lists, but implemented differently:
https://dzone.com/storage/temp/895349-arraylist-linkedlistt.png

*Any other methods that exist in LinkedList only?*

# Quiz time: List, ArrayList, and LinkedList

```
public static void main(String[] args) {

      List<String> lst1 = new ArrayList<String>();
      lst1.add("A");lst1.add("B");lst1.add("C");
      System.out.println(lst1.contains("A"));
      ((ArrayList<String>) lst1).ensureCapacity(100);
      System.out.println(lst1);

      List<String> lst2 = new LinkedList<String>();
      lst2.add("A");lst2.add("B");lst2.add("C");
      System.out.println(lst1.contains("A"));
      ((LinkedList<String>) lst2).addFirst("9");
      System.out.println(lst2);
}
```

Both AL and LL are lists, but implemented differently:
https://dzone.com/storage/temp/895349-arraylist-linkedlistt.png

*Any other classes implementing List?*

# A tour to source code of: List, ArrayList, and LinkedList

- http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/tip/src/share/classes/java/util/List.java
- http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/tip/src/share/classes/java/util/ArrayList.java
- http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/tip/src/share/classes/java/util/LinkedList.java

# Don't do this!

```
public interface GPSEnabled {
    public void printLocation();
}

public interface RadioEnabled {
    public int printLocation();
    public void startRadio();
    public void stopRadio();
}

public class Cellphone implements GPSEnabled, RadioEnabled {
    public void printLocation() {
      // return location
    }

    // ...

}
```

# Don't do this!

```
public interface GPSEnabled {
    public void printLocation();
}

public interface RadioEnabled {
    public int printLocation();
    public void startRadio();
    public void stopRadio();
}

public class Cellphone implements GPSEnabled, RadioEnabled {
    public void printLocation() {
      // return location
    }

    // ...
}
```

*What's wrong?*

*Name collision: Overlapped methods with different return types!*

# It's possible for an interface to extend interfaces

```
interface A { .. }
interface B { .. }

interface C extends A, B { .. }
```

# Interfaces for Cellphone: Say, you want to add a new method for interface RadioEnabled

```
public interface RadioEnabled {
    public void startRadio();
    public void stopRadio();

}

public class Cellphone implements RadioEnabled {
    public void startRadio() {
      // start radio
    }
    public void stopRadio() {
      // stop radio
    }
}
```

# Interfaces for Cellphone: Say, you want to add a new method for interface RadioEnabled

```
public interface RadioEnabled {
    public void startRadio();
    public void stopRadio();
    public void recordRadio();
}

public class Cellphone implements RadioEnabled {
    public void startRadio() {
      // start radio
    }
    public void stopRadio() {
      // stop radio
    }
}
```

*What might happen?*

# Interfaces for Cellphone: Say, you want to add a new method for interface RadioEnabled

```
public interface RadioEnabled {
    public void startRadio();
    public void stopRadio();
    default public void recordRadio() {
       System.out.println("Recording radio..");
    }
}

public class Cellphone implements RadioEnabled {
    public void startRadio() {
      // start radio
    }
    public void stopRadio() {
      // stop radio
    }
}
```

***We can have a default implementation of a method in an interface (Java 8+)***

# Interfaces for Cellphone: Say, you want to add a new method for interface RadioEnabled

```
public interface RadioEnabled {
    public void startRadio();
    public void stopRadio();
    default public void recordRadio() {
       System.out.println("Recording radio..");
    }
}

public class Cellphone implements RadioEnabled {
    public void startRadio() {
      // start radio
    }
    public void stopRadio() {
      // stop radio
    }
}
```

***We can have a default implementation of a method in an interface (Java 8+)***

***So the new method won't break your existing code!***

Quiz time: Have a look at Comparable interface in JavaDoc, and try to create an Employee class implementing the Comparable interface, such that employees can be compared based on the order of their instantiations (employees created first get more priorities than those created later).

From your implementation, demonstrate how you can sort an Employee list, based on the employee instantiation order.


**Inspired by:**

https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html
Liang. Introduction to Java Programming. Tenth Edition. Pearson 2015.
Think Java book by Allen Downey and Chris Mayfield.
Eck. Introduction to Programming Using Java. 2014.