Abstract Class & Interface

An abstract class is a class that is declared abstract —it may or may not include abstract methods.

An interface is just like Java Class, but it only has static constants and abstract method.

Both abstract classes and interfaces cannot be instantiated.

Abstract Classes

Abstract Classes

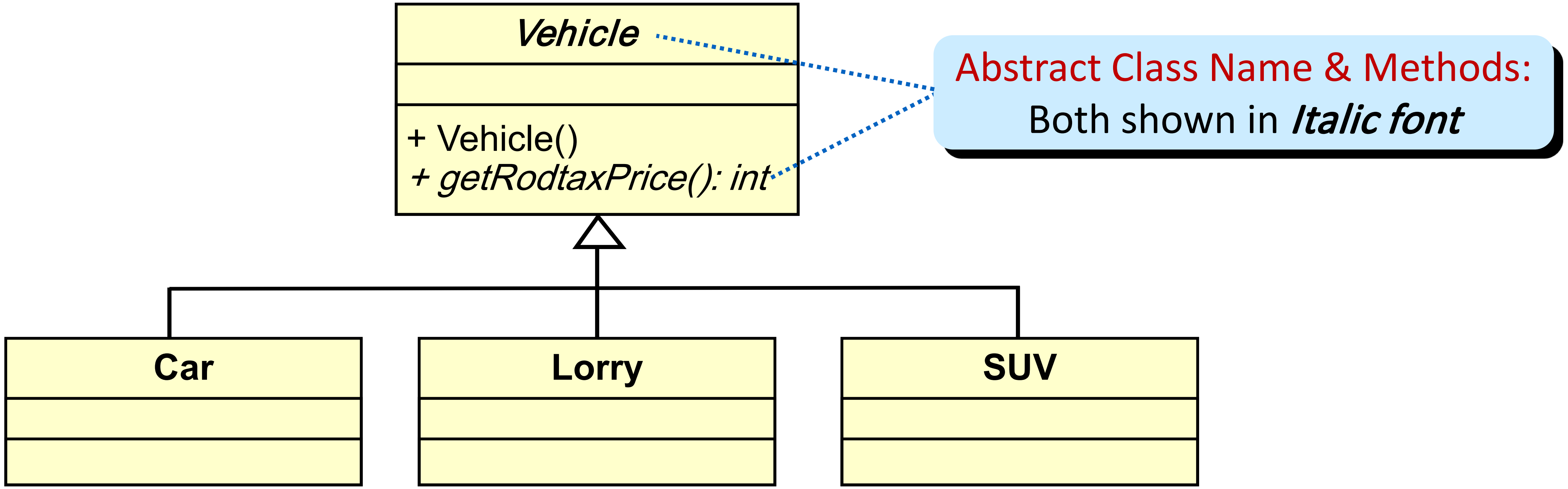
- An abstract class **cannot be instantiated**, but other classes are derived from it, or **extended**. It serves as a **superclass for other classes**.
- The abstract class **represents the generic** or abstract form of all the classes that are derived from it.
- A class becomes abstract when you place the **`abstract`** keyword in the class definition.
- An abstract class often **contains abstract methods**, though it doesn't have to.

Abstract Classes vs. Concrete classes

- **Concrete** means that the class **can have instances**. In contrast, **abstract** means that a class **cannot have its own instances**.
- **Abstract classes** exist purely to **generalize common behavior/method** that would otherwise be **duplicated across (sub or concrete) classes.**
- **Concrete Classes** are regular classes, where all **methods are completely implemented**.
- When an abstract class is **subclassed**, the subclass usually **provides implementations** for all **of the abstract methods** in its **abstract (parent) class**.

Abstract Classes in UML Representation

- Abstract class **doesn't support multiple inheritance** similar to inheritance which doesn't allow multiple inheritance.


Vehicle
+ Vehicle()
+ getRodtaxPrice(): int
Abstract Class Name & Methods:
Both shown in *Italic font*
Car
Lorry
SUV

Why Abstract Classes?

- It helps **establish common elements** in a class that is too general to instantiate.
- Sometimes we don't want objects of a base class to be created.
- Example
  - *Vehicle*: **Car, Lorry, SUV , …** ← **What does a generic *Vehicle* look like?**
  - *Animal*: **Cat, Fish, Lion, …** ← **An *Animal* object makes no sense** Example

→ **Solution:** make it an abstract class – cannot be instantiate.

What Inside Abstract Classes?

- **Fields** – that are **not-static** and **not-final**. This enables you to define methods that can access and modify the state of the object to which they belong.
- **Constructor** – in Java, constructor doesn't actually "build" the object, it is used to **initialize fields**.
- **Methods** – both; **fully implemented methods** or **abstract methods**.

`abstract` Keyword: to Declare Abstract Classes

- use the modifier **`abstract`** on a class header to declare an abstract class

```
abstract public class Shape {...}
               Vs.
     public class Shape {...}
```

Abstract Methods

- An abstract method has **only a header** and **no body (statement or implementation).**
- An abstract method has **no body** and **must be overridden in a subclass.**
- Any **class** that **contains an abstract method** is **automatically abstract.**
- If a **subclass fails to override** an abstract method, a **compiler error will result.**
- Abstract methods are used to ensure that a subclass **implements** the methods.

`abstract` Keyword: to Declare Abstract Methods

- The abstract methods *have no bodies*, only headers that are *terminated by semicolons* (;).

```
abstract public int getArea();
Vs.
public int getArea(){return area;}
```

Abstract Class & Method Codes Example

Abstract Class

```java
abstract public class Shape {
    public String color;
    public Shape(String color){
        this.color = color;
    }
    abstract public String toString();
    abstract public double getArea();
    public double getColor(){
         return color;
    }
}
```

Abstract Clases

Modifier **abstract** is used to declare an abstract class and its placed in the class header.

Abstract Methods

Notice that the keyword **abstract** appears in the header, and that the header ends with a **semicolon**.

Abstract Class & Method Codes Example

Concrete Class

```
class Rectangle extends Shape {

    public int length, width;

    public Rectangle(String color, int length, int width){
        super(color);
        this.length = length;
        this.width = width;
    }

    @Override
    public String toString(){
        return "Rectangle[length=" + length + ",width=" + width + "]";
    }

    @Override
    public double getArea(){
        return length*width; }

    public double getWidth(){
        return this.width; }
}
```

Abstract Methods

Abstract method is **declared** in a **superclass**, but **implemented** or *overridden* in a **subclass**.

Abstract Class & Method Codes Example

TestShape.java – Main

```
public class TestShape {

    public Static void main(String[] args){

        // Parent class reference to child object - polymorphism
        Shape s1 = new Rectangle("red", 4, 5);
        System.out.println(s1);
        System.out.println("Area is " + s1.getArea());

        // Cannot create instance of an abstract class
        Shape s3 = new Shape("green");    // Compilation Error!!

    }
}
```

[Output]

```
Rectangle[length=4,width=5,Shape[color=red]]
Area is 20.0
```

Summary of Abstract Classes

- An abstract class is incomplete in its definition, since the implementation of its abstract methods is missing. Therefore, an abstract class cannot be instantiated. In other words, you cannot create instances from an abstract class.
- To use an abstract class, you have to derive a subclass from the abstract class. In the derived subclass, you have to override the abstract methods and provide implementation to all the abstract methods. The subclass derived is now complete, and can be instantiated.

Interface

Interface

- An interface is **similar to an abstract class** that has **all abstract methods.**
  - It **cannot be instantiated**, and
  - methods listed in an interface must be **implemented in other classes**.
- The **purpose** of an interface is to **specify behavior** for other classes.
- An **interface** looks **similar** to a **class**, **except**:
  - the keyword **`interface`** is used instead of the keyword ~~`class`~~, and
  - the methods that are specified in an interface **have no bodies**, **only headers** that are **terminated by semicolons**.

Declaration of Interface

- The keyword `interface` is used instead of the keyword `class`.

```
public Interface Shape {...}
            vs.
   public class Shape {...}
```

Interface Implementation

- A class can **implement** one or **more interfaces** – *multiple inheritance.*
- If a class implements an interface, it uses the **`implements`** keyword in the class header.

```
public class Rectangle extends Shape
implements Contour, Thing {...}
```

OR

```
public class Rectangle implements
Contour, Thing {...}
```

Methods in Interface

- All methods specified by an interface are **public by default**.
- It is necessary that the class must **implement all the methods declared in the interfaces** and it should **override all the abstract methods declared in the interface to provide implementation.**
- The methods *have no bodies*, only headers that are *terminated by semicolons* (;).

```
abstract public int getArea();
```

Fields in Interfaces

- An interface can contain field declarations:
  - **all fields in an interface are treated** as `final` and `static`.
- Because they **automatically become →** `final`, you must **provide an initialization value**.

```
    public interface Contour{
          int FIELD1 = 1, FIELD2 = 2;
          (Method headers...)
    }
```

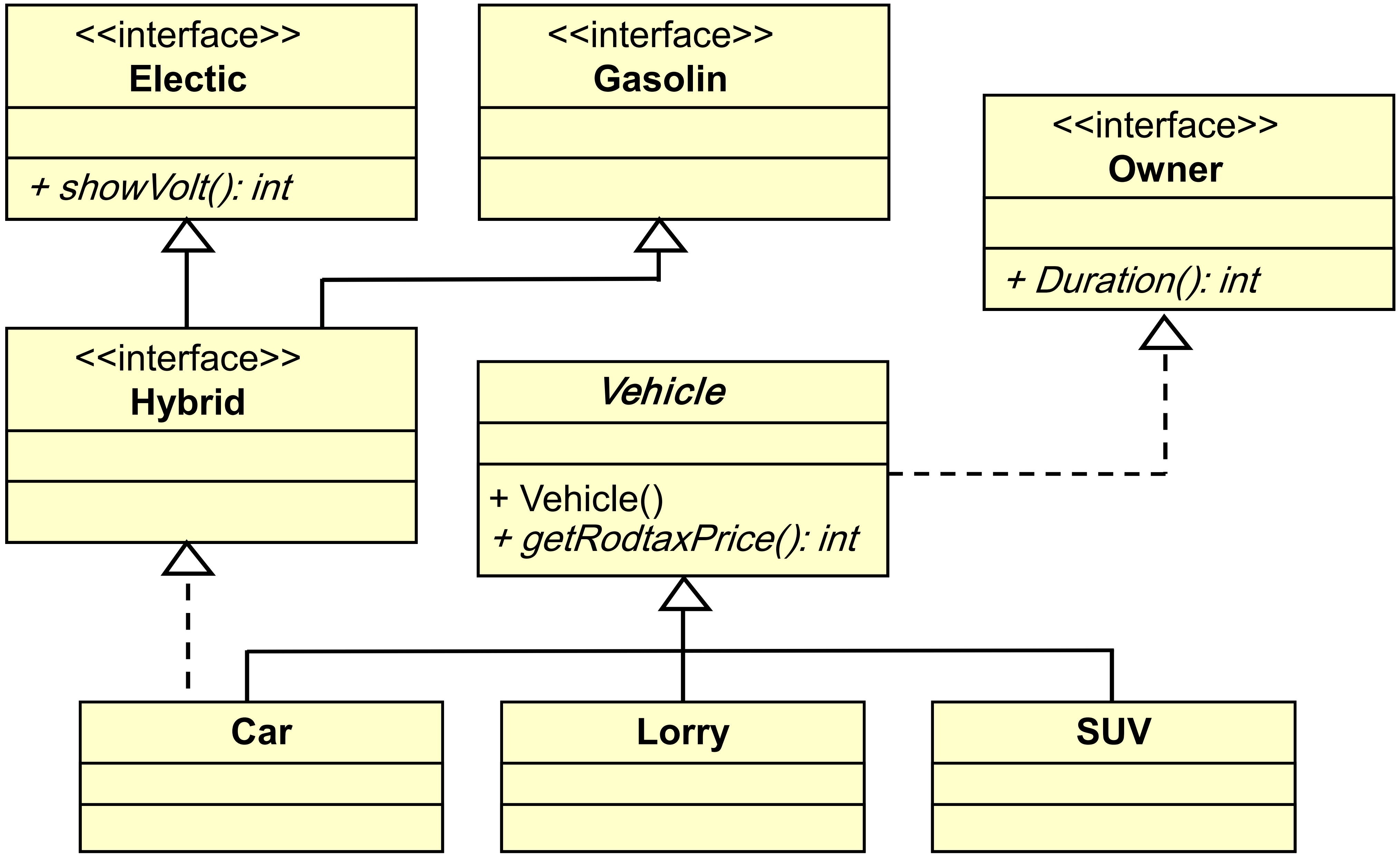
- In this interface, `FIELD1` and `FIELD2` are `final static int` variables.
- Any class that implements this interface **has access to these variables**.

Implementing Multiple Interfaces

- **Normally** in Java **a class** can be **derived from** only **one superclass** → **Inheritance**. But **Java allows** a class to **implement multiple interfaces**.
- When a class **implements** multiple interfaces, **it MUST provide the methods specified by ALL of them**.
- To specify multiple interfaces in a class definition, simply **list the names of the interfaces**, **separated by commas**, after the **`implements`** key word.

```
public class Class1 implements  Interface1,
                                Interface2,
                                Interface3
```

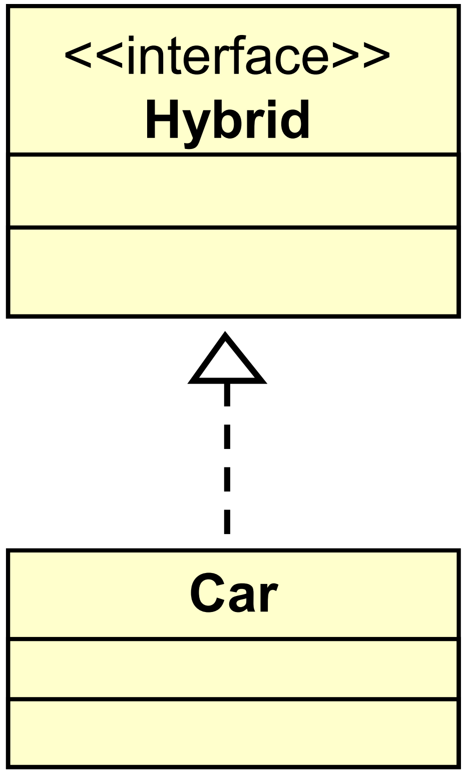
Interface in UML Representation



Polymorphism with Interfaces

- Java **allows** you to create **reference variables of an interface type.**
- An **interface reference variable can reference any object that implements that interface**, regardless of its class type.
- When a **class implements an interface**, an inheritance relationship known as *interface inheritance* is established.
- When an interface variable references an object:
  - **only the methods declared in the interface are available**,
  - **Downcasting** is required to access the other methods of an object referenced by an interface reference.

Polymorphism with Interfaces



- When an interface variable references an object:
  - **only the methods declared in the interface are available**,
  - **Downcasting** is required to access the other methods of an object referenced by an interface reference.
- You cannot create an instance of an interface.

```
Hybrid car = new Hybrid(); // error!!

Hybrid car = new Car(); // OK!!
```

Interface Codes Example

Interface Hybrid

```
public interface Electric {

    String POWERSOURCE = "Electic";


    abstract void showPowerSource();
    }
```

Class Car

```
public class Car implements Electric, Hybrid{
    @Override
    public void showPowerSource(){
        System.out.println("Battery Powered");
    }

    @Override
    public void showPowertrain(){
        System.out.println("Hybrid");
    }

    public void move(){
        System.out.println("vrom..vrom...");
    }
}
```

Interface Hybrid

```
interface Hybrid {

    String POWERSOURCE = "Electric & Gasolin";


    abstract void showPowertrain();
    }
```

Interface Codes Example – main() class

```java
public class TestInterface {

    public static void main(String args[]){
        Hybrid car1 = new Car();
        car1.showPowertrain();
        System.out.println(Hybrid.POWERSOURCE);
        car1.move(); // ERROR!! - interface reference variable
        ((Car) car1).move(); //downcasting


        Electric car2 = new Car();
        car2.showPowerSource();
        System.out.println(Electric.POWERSOURCE);

        Electric car3 = new Electric(); // ERROR!! - interface instantiation
    }
}
```

[Output]

```
Hybrid
Electic & Gasolin
vrom..vrom...
Battery Powered
Electic
```