Object-Oriented Programming

Abstract Methods and Classes
Interfaces

United International College

Abstract Methods

- Some methods should not have code.
  - Example: the **`area`** method of the **`Shape`** class, because we cannot compute the area of some unknown shape.

```
public class Shape {
    ...
    public double area() {
        System.out.println("Unknown area!");
        return -1.0; // We must return a double.
    }
}
```

- So make the methods **`abstract`**: the methods have no code, only a `;`

```
public abstract double area();
```

Abstract Methods

- Non-private abstract methods are inherited by subclasses, just like other methods.
- Abstract methods will later be overridden in subclasses using non-abstract methods, just like other methods.
  - Example: the `Circle` class overrides the abstract `area` method inherited from the `Shape` class, to compute the area of a circle using `Math.PI`.
- It is possible to override an inherited abstract method with another abstract method, but it is useless since it just replaces one abstract method with another.

Abstract Classes

- If a class contains at least one abstract method then the class is missing some code.
- Therefore it is not possible to create objects from this class (because such objects would be missing some code!)
- To indicate this, the class itself must be marked as `abstract`.

Abstract Classes

- A class which inherits abstract methods from a superclass, and does not override all the inherited abstract methods, is missing code too. Therefore is must be marked as `abstract` too.

- A class which inherits abstract methods from a superclass, and does override all the inherited abstract method, is not missing any code and can be used just like any other non-abstract class.

Example

```
public abstract class Shape {
        ...
        public abstract double area();
}
...
public class Circle extends Shape {
        ...
        @Override
        public double area() {
                return Math.PI * radius * radius;
        }
}
```

Example

```
public abstract class BigShape extends Shape {
        ...
        @Override
        public abstract double area(); // Useless
}
...
public class Test {
        public static void main(String[] args) {
                Shape s = new Shape();   // Forbidden
                Circle c = new Circle(); // Okay
                ...
        }
}
```

Abstract Classes

- An abstract class can only be extended by subclasses, that is the only thing you can do with it.
- The abstract class can still contain constructors, as well as instance variables and non-abstract methods which are inherited by subclasses.
- It is possible to write an abstract class that contain only abstract methods, but this is rare. In that case the class is only useful for containing instance variables and constructors.

Interfaces

An interface is a specification of what some classes can do (which public methods they have) without saying how the classes do it (which code the methods have).

- Example: an interface called **`Movable`** with **`getSpeed`** and **`setSpeed`** methods, for all the classes that have objects that can move at different speeds.
- Example: an interface called **`Edible`** with a **`cook`** method, for all the classes that have objects that can be cooked.
- All methods listed in an interface are always **`public`**.

Example

```
public interface Movable {
	public double getSpeed();
	public void setSpeed(double newSpeed);
}

public interface Edible {
	public void cook();
}
```

Interfaces

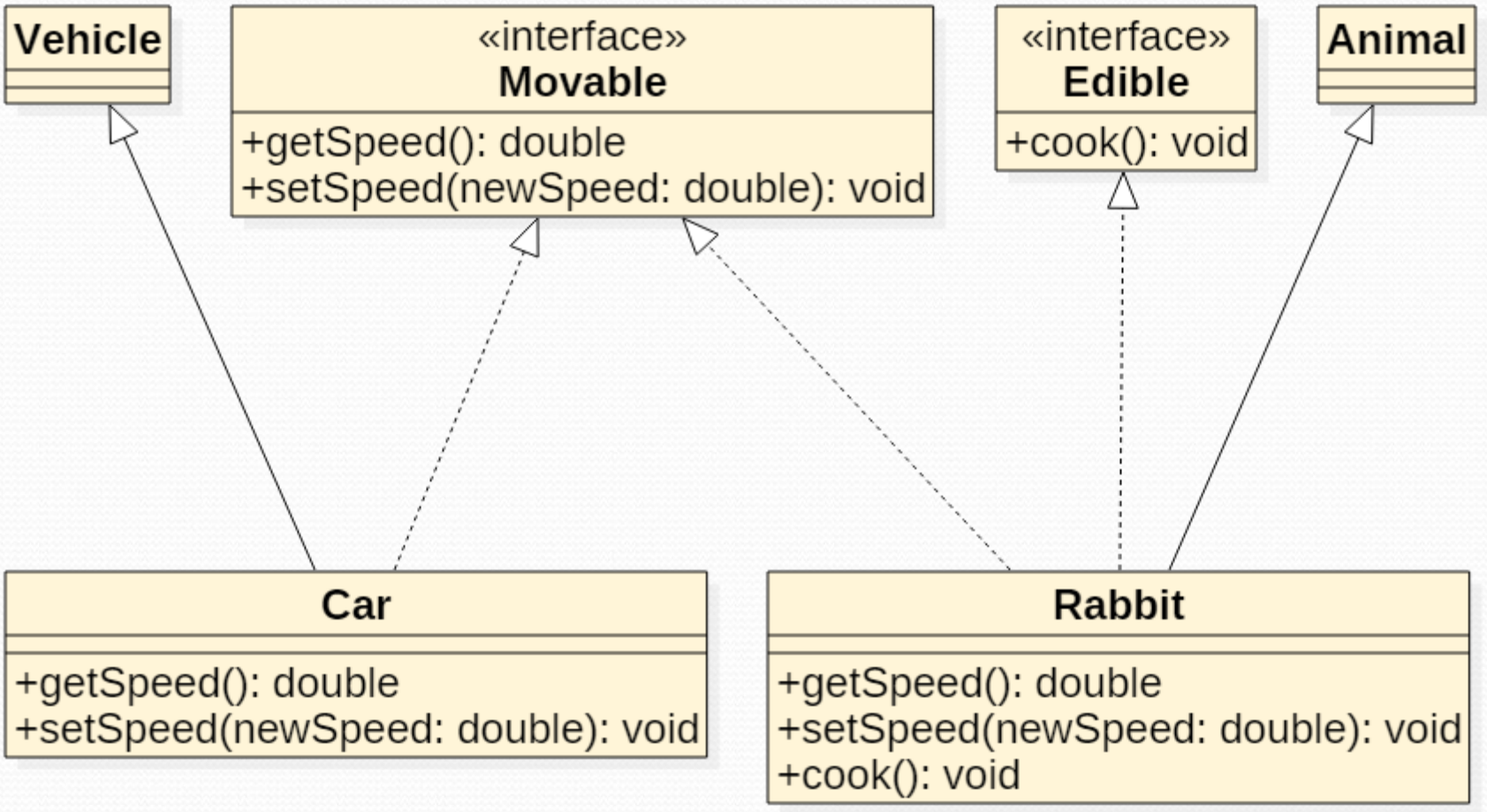
- Multiple unrelated classes can implement the same interface.
  - Example: a **`Car`** class and a **`Rabbit`** class both might have **`getSpeed`** and **`setSpeed`** methods to implement the **`Movable`** interface, even though the two classes are unrelated (their only common ancestor is the **`Object`** class).
- The same class can implement multiple unrelated interfaces.
  - Example: a **`Rabbit`** class might implement both the **`Movable`** interface and the **`Edible`** interface.

Example

```
public class Car extends Vehicle implements Movable {
        ...
        @Override // Asks Java to check for "implements" errors.
        public double getSpeed() { ... }
        @Override
        public void setSpeed(double newSpeed) { ... }
}

public class Rabbit extends Animal implements Movable, Edible {
        ...
        @Override
        public double getSpeed() { ... }
        @Override
        public void setSpeed(double newSpeed) { ... }
        @Override
        public void cook() { ... }
}
```

Example


Vehicle
«interface»
Movable
+getSpeed(): double
+setSpeed(newSpeed: double): void
«interface»
Edible
+cook(): void
Animal
Car
+getSpeed(): double
+setSpeed(newSpeed: double): void
Rabbit
+getSpeed(): double
+setSpeed(newSpeed: double): void
+cook(): void

Interfaces

- An interface is specified using the **`interface`** keyword.
- An interface never contains any code! It describes what can be done (method names), not how to do it (method code)! Therefore it is not possible to create an object from an interface!
- A class specifies that it implements an interface using the **`implements`** keyword.
- The class must then provide code for all the methods listed in that interface (except if the class is abstract, in which case some of the methods listed in the interface can be abstract in the class).

- An interface can also contain constants (**`static final`** data) but this is more rare.

Interfaces

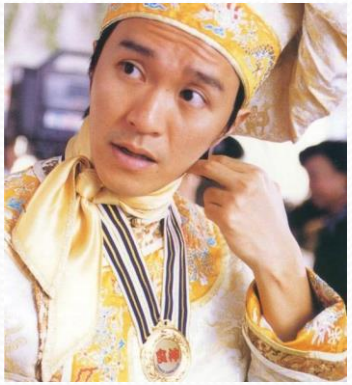
implements

implements

Stephen Chow

implements

implements



Interfaces

- An interface splits your software into two parts:
  - The classes that implement the interface (provide code for all the methods listed in the interface).
  - The rest of the code that uses the interface (call the methods listed in the interface).
- The classes that implement the interface do not need to know anything about how the rest of the code uses the interface.
- The rest of the code that uses the interface does not need to know anything about how the classes implementing the interface work internally.

Interfaces

- The interface creates a contract between the two sides:
  - The implementation side promises to provide code for all the methods listed in the interface.
  - The use side promises to use only the methods listed in the interface.
- The two sides can then be developed by different teams of software engineers independently of each other.
- The interface becomes the Application Programming Interface (API) between the two sides.

Interfaces

- A building architect splits a building into separate rooms with doors in precise places that allow the different rooms to communicate with each other in specific ways. Each room can then be decorated by a different team of decorators.

- A software architect splits a software into separate components with interfaces in precise places that allow the different components to communicate with each other in specific ways. Each component can then be developed by a different team of software engineers.

Interfaces

- Just like a class, an interface can also be used as a type.
- Example: the **`Rabbit`** class implements the **`Edible`** interface, therefore a rabbit object can be assigned to a variable of type **`Edible`**:

```
Edible e = new Rabbit();
e.cook(); // Cooking the rabbit.
```

- This is the same subtyping polymorphism we have seen when studying class inheritance (a rabbit object can be used as an animal object if the **`Rabbit`** class is a subclass of the **`Animal`** class) but with interfaces.

Interfaces

- If a class implements an interface, then any subclass implicitly implements the same interface too, because the subclass inherits from its parent class all the methods necessary to implement the interface.

- Example:

```
public class Car extends Vehicle implements Movable { ... }
public class Taxi extends Car { ... }
Movable m = new Taxi();
```

- The **`Car`** and **`Taxi`** classes both implement the **`Movable`** interface, the **`Vehicle`** class does not.

Interfaces

- Using interfaces, you can write code that processes data without knowing what kind of data it really is.
- Example:

```
public class Test {
    ...
    public void stopMovable(Movable m) {
        m.setSpeed(0.0); // m's speed is now zero.
    }
}
```

- The **`stopMovable`** method stops a movable object, but it does not know whether the movable object is a car or a rabbit. It only knows that the object is movable (which is really the only thing it needs to know).

Interfaces

When to use interfaces?

- Any time you want to create a new API inside your software, to split it into two parts (the API implementation part and the API use part). The API implementation part can then be used throughout the rest of the software while being independent of it.
- Any time you have multiple people working on the same software and you want to have each person be able to work on their own part independently of the other people.

Interfaces

If multiple unrelated classes contain the same methods, make it an interface.

- If the unrelated **`Car`** and **`Rabbit`** classes both have **`getSpeed`** and **`setSpeed`** methods then create a **`Movable`** interface for the two methods.
- If the related **`Student`** and **`Teacher`** classes both have **`getName`** and **`setName`** methods then move the methods into the common superclass **`Person`**. (The **`Person`** class can later implement a **`Nameable`** interface for the two methods, if you want.)

Interfaces vs. Abstract Classes

What is the difference between an interface and an abstract class which only has abstract methods?

- Both contain no code.
- Both cannot be used to create objects.
- Both provide subtyping polymorphism.

Interfaces vs. Abstract Classes

- The interface splits your code into two parts and works as an API between the two parts. The classes implementing the interface are unrelated to each other (`Car` and `Rabbit`), they just happen to be able to do some of the same things.

- The abstract class is used for code inheritance (even if all the methods are abstract and no code is actually inherited). The subclasses are closely related to each other (`Student` and `Teacher`) since they are special cases of the same parent class (`Person`).

Interfaces vs. Abstract Classes

	Abstract class	Interface
Is it a Class?	Yes	No
Create Ojbect	No	No
Relationship with Class	Inheritance	A class can implement multiple interfaces
Members	May have private memebers	Default to pulic static final，any class implement the interface has the access to these constant.
Methods	May be private，non-abstract which has implementation	Default to public，abstract
Design	"is-a" relationship	"like-a"relationship
Implementation	extends	implements
Derived/Concrete Methods	Implement abstract methods or still be abstract	Must implement all methods in the interface

Exercise

- Create an **`Animal`** class with a boolean **`canFly()`** method. Should the class be abstract or not?
- Create a subclass **`Cow`** of **`Animal`**. Should the class be abstract or not?
- Create a subclass **`Bat`** of **`Animal`**. Should the class be abstract or not?
- Create a class **`Airplane`**, which implements a **`Flyable`** interface that has two methods **`takeOff()`** and **`land()`**.
- Modify the **`Bat`** class to implement **`Flyable`** too.
- Add a **`Test`** class with a method **`makeItTakeOff(Flyable f)`** that makes the flyable object **`f`** take off.

Summary

- Abstract Methods and Classes
- Interfaces