Lazy Load

Java & OOP

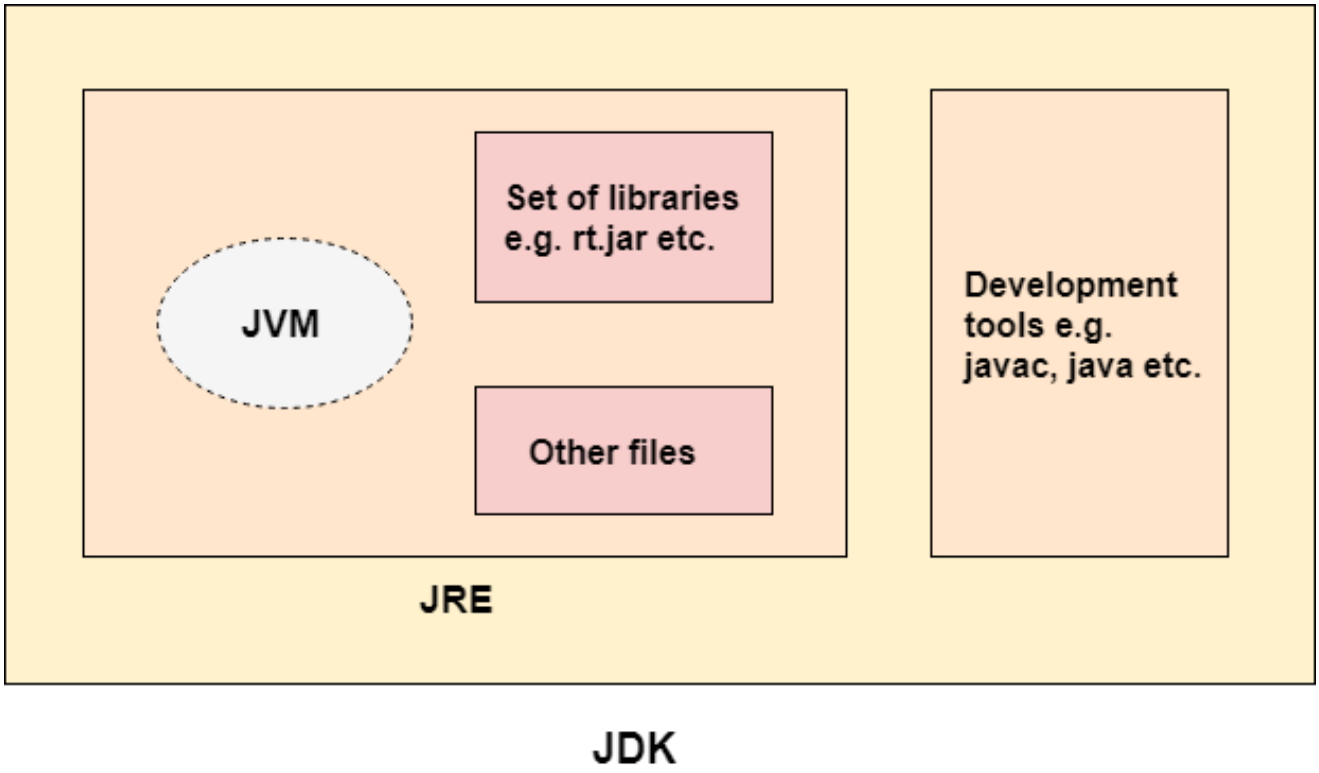
Repo

https://github.com/TAIsRich/chuwa-eij-tutorial.git

JAVA Basic

JDK vs. JRE vs. JVM

The **JDK** contains a private **Java Virtual Machine (JVM)** and a few other resources such as **an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc**. to complete the development of a Java Application.



Varibale Types

Primitive types(原始类型数据, 能用 “==” 比较)

`byte, short, int, long, float, double, char, boolean`

Data Type	Size	Description
`byte`	1 byte	Stores whole numbers from -128 to 127
`short`	2 bytes	Stores whole numbers from -32,768 to 32,767
`int`	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
`long`	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`float`	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
`double`	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
`boolean`	1 bit	Stores true or false values (1 byte = 8 bit, In binary 0,1)
`char`	2 bytes	Stores a single character/letter or ASCII values

e.g.

```
int myNum = 5;               // Integer (whole number)
float myFloatNum = 5.99f;    // Floating point number
char myLetter = 'D';         // Character
boolean myBool = true;       // Boolean
String myText = "Hello";     // String
```

Refrence types(引用类型数据)

`class, interface, array` 本质都用到了类似于C语言的指针

```
public int add(int a, int b){
  return a + b;
}

public int changeAge(User u1, User u2) {
  u1.setAge(18);
  return u1.age + u2.age;
}
```

```
9
10 public static void main(String[] args) {
11   User u1 = new User(20);
12   u1 = new user(30);
13   User u2 = new User(22);
14   changeAge(u1, u2);
15   u1.getAge()  //  -> 18
16 }
17
```

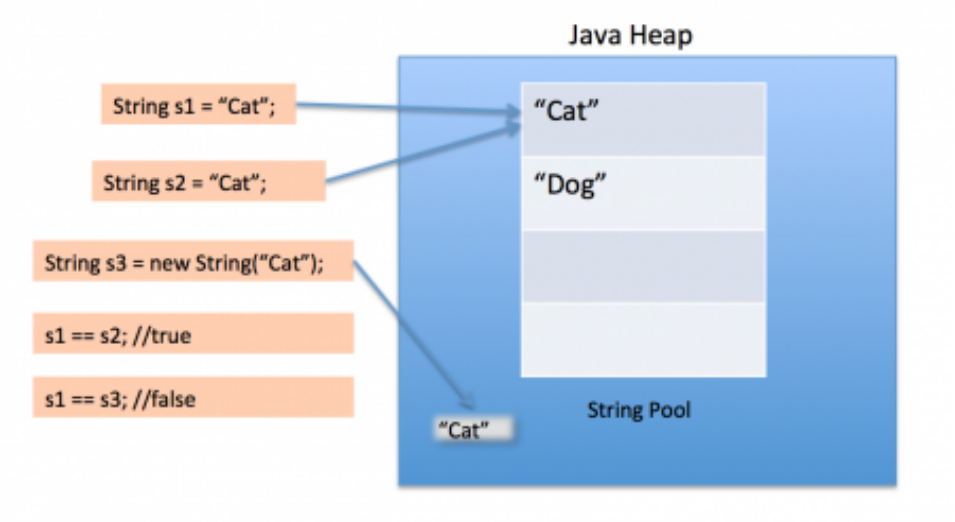
Questions: what is passing by value or passing by Ref ?

https://www.cs.virginia.edu/~jh2jf/courses/cs2110/java-pass-by-value.html

String Constant Pool(字符串常量池)

Question: What is difference between `String a = "Cat"` and `String b = new String("Cat")`?

Note: String is immutable



Question: Why java string is **immutable** ?

Final

Variable

- `public static final String APP_NAME="testApp"`
- **Purpose**: define **constants**

Method

- `public final int add(int a, int b){ return a + b; }`
- **Purpose**: prevent **override**

Class

- `final class MyClass(){}`
- **Purpose**:
    - 1. prevent **inheritance**, like Integer, String etc;
      2. Make class **immutable**
-

Interview Questions:
difference between: final, finally, finalize

Static

Static Variable and Static Block

only one Instance

```
public class EmployeeRepository {
    private static Set<Employee> employees = new HashSet<>();

    static {
        employees.add(new Employee( id: 1001,  name: "Yun Ma",  age: 50,  salary: 30000.00));
        employees.add(new Employee( id: 1002,  name: "Huateng Ma",  age: 49,  salary: 22222.22));
    }

    public Set<Employee> getEmployees() {
        return employees;
    }
}
```

Static Method

Can directly call static method using Class name

```
public class EmployeeRepository {
    private static Set<Employee> employees = new HashSet<>();

    static {
        // static method: EmployeeData.getEmployees()
        employees.addAll(EmployeeData.getEmployees());
    }

    public Set<Employee> getEmployees() {
        return employees;
    }
}
```

```
public class EmployeeData {

    public static List<Employee> getEmployees() {
        List<Employee> list = new ArrayList<>();

        list.add(new Employee( id: 1001,  name: "Yun Ma",  age: 50,  salary: 30000.00));
        list.add(new Employee( id: 1002,  name: "Huateng Ma",  age: 49,  salary: 22222.22));
        list.add(new Employee( id: 1004,  name: "Jun Lei",  age: 43,  salary: 12234.12));
        list.add(new Employee( id: 1005,  name: "Bill Gates",  age: 65,  salary: 999999));
        list.add(new Employee( id: 1003,  name: "Yanhong Li",  age: 30,  salary: 123123));
        list.add(new Employee( id: 1007,  name: "Zhengfei Ren",  age: 78,  salary: 66666));
        list.add(new Employee( id: 1006,  name: "Mark Elliot Zuckerberg",  age: 29,  salary: 88888));

        return list;
    }
}
```

Questions:

1. Can static method access non-static varibales ?
2. Common Static Methods ?
3. When to use static methods ?
    1. 工具类的方法一般都设计成static。
    2. Integer, String, Math, System etc.
4. How to call static methods?

Static Class

```
import org.junit.jupiter.api.Test;
```

```java
/**
 * @author b1go
 * @date 6/7/22 10:47 PM
 */
public class Client {
    @Test
    public void testStaticClass() {
        CarParts.StaticWheel staticWheel = new CarParts.StaticWheel();
        staticWheel.drive();
        CarParts.combine();
        CarParts.NonStaticWheel nonStaticWheel = new CarParts().new NonStaticWheel();
        nonStaticWheel.toString();
    }
}
```

```java
/**
 * @author b1go
 * @date 6/7/22 10:43 PM
 */
public class CarParts {
    // static wheel class
    public static class StaticWheel {
        public StaticWheel() { System.out.println("Static Wheel created"); }

        public void drive() { System.out.println("drive static wheel"); }
    }

    // non static wheel class
    public class NonStaticWheel {
        public NonStaticWheel() { System.out.println(" Non Static Wheel Created"); }
    }

    // default class
    public CarParts() { System.out.println("Car parts Object Created!"); }

    public static void combine() { System.out.println("combine car parts"); }
}
```

JVM Load

static variable/block -> constructor(used in Obj)

```java
public class JvmLoad {
    public static void main(String[] args) {
        Demo demo = new Demo();
    }
}

class Demo {
    // default value is 0
    private static int n1;
    private static final String s1 = "static variable";
    private static Set<Employee> employees = new HashSet<>();
    private String s2 = "non static variable";

    static {
        System.out.println("******* 1, static block is called ***** ");
        System.out.println("******* 2, check values of static variables  ***** ");
        System.out.println(n1);
        System.out.println(s1);
        System.out.println(employees);
        System.out.println("**** done ****\n");
        // static method: EmployeeData.getEmployees()
        employees.addAll(EmployeeData.getEmployees());

        // in static block, call non-static variable
//        System.out.println(s2);
    }

    public Demo() {
        System.out.println("******* 3, Constructor is called ******");
        System.out.println(n1);
        System.out.println(s1);
        employees.forEach(System.out::println);
        System.out.println("---- s2 have value now ------");
        System.out.println(s2);
        System.out.println("*** done *****");
    }
}
```

```
******* 1, static block is called *****
******* 2, check values of static variables  *****
0
static variable
[]
**** done ****

******* 3, Constructor is called ******
0
static variable
Employee{id=1004, name='Jun Lei', age=43, salary=12234.12}
Employee{id=1003, name='Yanhong Li', age=30, salary=123123.0}
Employee{id=1006, name='Mark Elliot Zuckerberg', age=29, salary=88888.0}
Employee{id=1002, name='Huateng Ma', age=49, salary=22222.22}
Employee{id=1005, name='Bill Gates', age=65, salary=999999.0}
Employee{id=1001, name='Yun Ma', age=50, salary=30000.0}
Employee{id=1007, name='Zhengfei Ren', age=78, salary=66666.0}
---- s2 have value now ------
non static variable
*** done *****

Process finished with exit code 0
```

debug to show the load sequences

```
class Demo {
    // default value is 0
    private static int n1;   n1: 0
    private static final String s1 = "static variable";   s1: "static variable"
    private static Set<Employee> employees = new HashSet<>();   employees:  size = 0
    private String s2 = "non static variable";

    static {
        System.out.println("******* 1, static block is called ***** ");
        System.out.println("******* 2, check values of static variables  ***** ");
        System.out.println(n1);
        System.out.println(s1);
        System.out.println(employees);
        System.out.println("**** done ****\n");
        // static method: EmployeeData.getEmployees()
        employees.addAll(EmployeeData.getEmployees());
```

Comment

1. `/* text */
    1. The compiler ignores everything from /* to */.
2. // text
    1. The compiler ignores everything from // to the end of the line.
3. /** text */
    1. This is a documentation comment and in general its called doc comment. The JDK javadoc tool uses doc comments when preparing automatically generated documentation.

Java commands

```
$> javac HelloWorld.java
$> java HelloWorld(.class)
$> javadoc -d HelloWorld HelloWorld.java
```

OOP

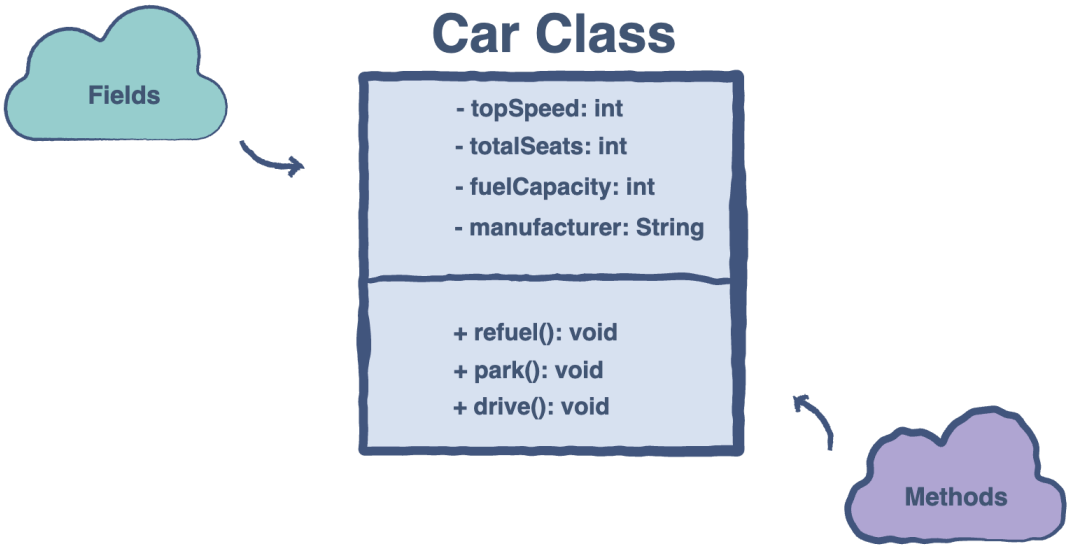
Class and Objects

Question: what is **class** ? what is **object** ?

An **object** is an instance of a module, and a **class** is its definition. (e.g. "Human" class --> "Bob" object)

Components

- Fields
- Methods
- Constructor



Constructor

- Default constructor
  - `public Car() {}`
  - 如果我们没有写，JVM会自动帮我们提供该constructor
- Parameterized constructor
  - `public Car(Sting brand, String carType, Wheel wheel){...}`
  - 如果我们写了有参constructor, JVM 就不会帮我们提供default constructor, 如果我们需要，则需要自己写。

Calling a constructor from a constructor

Calling a constructor from a constructor

Quetions: what does **this** keyword mean ? how about **super** ?

Code Demo: ConstructorLearn

Encapsulation(封装)

- **Getter/Setter** (methods)

Encapsulation in OOP refers to binding the **data** and the **methods to manipulate that data** together in a single **unit** (class).

```java
/* File name : EncapTest.java */
public class EncapTest {
   private String name;
   private String idNum;
   private int age;

   public int getAge() {
      return age;
   }

   public String getName() {
      return name;
   }

   public String getIdNum() {
      return idNum;
   }

   public void setAge( int newAge) {
      age = newAge;
   }

   public void setName(String newName) {
      name = newName;
   }

   public void setIdNum( String newId) {
      idNum = newId;
   }
}
```

```java
/* File name : RunEncap.java */
public class RunEncap {

   public static void main(String args[]) {
      EncapTest encap = new EncapTest();
      encap.setName("James");
      encap.setAge(20);
      encap.setIdNum("12343ms");

      System.out.print("Name : " + encap.getName() + " Age : " + encap.getAge());
   }
}
```

Tips: use getter/setter instead of direct access

Industry standard code structure

Check code in oop

Access Modifiers

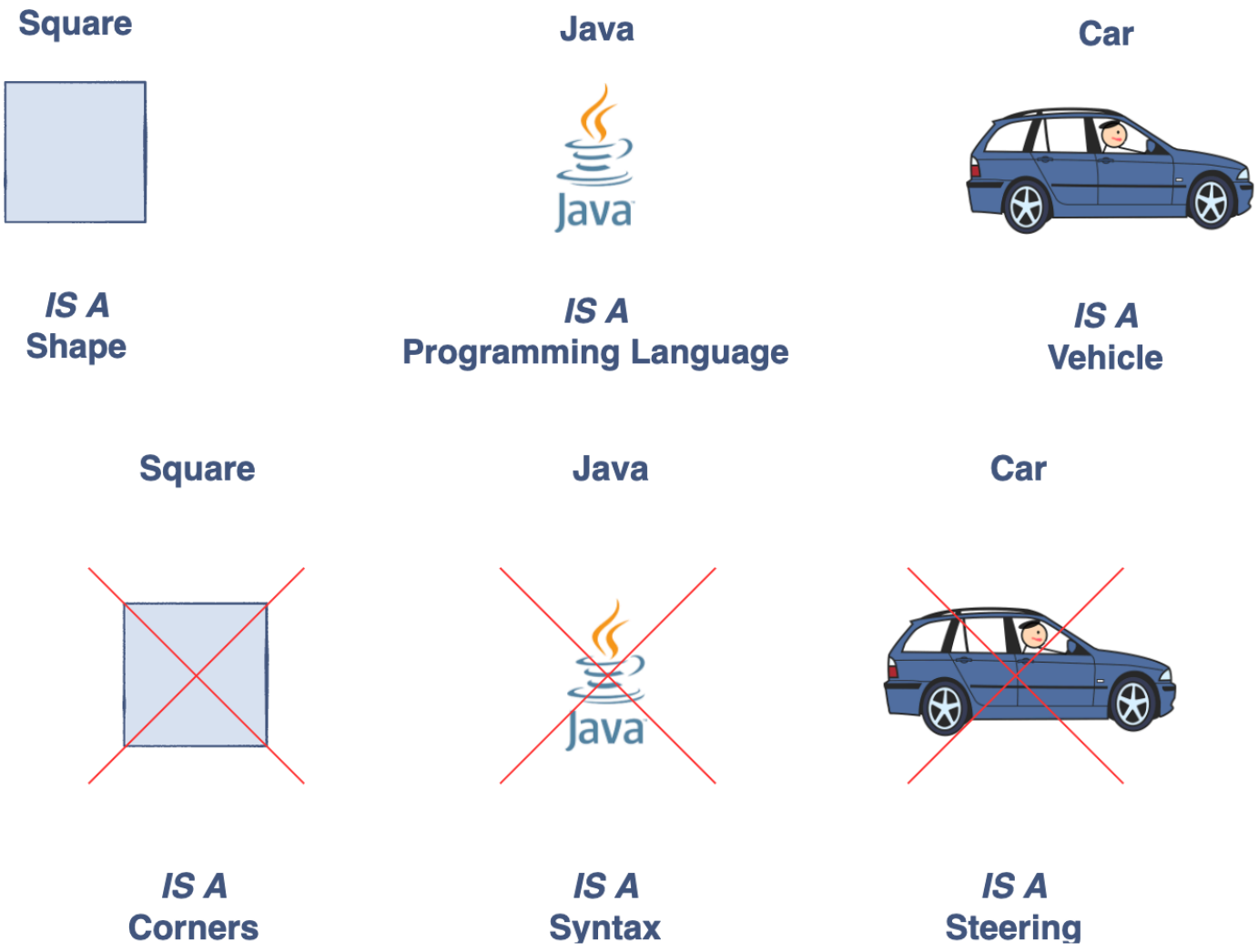
Modifier	Description
Default	declarations are visible only **within the package (package private)**
Private	declarations are visible **within the class only**
Protected	declarations are visible **within the package or all subclasses**
Public	declarations are visible **everywhere**

Inheritance(继承)

Inheritance(继承)

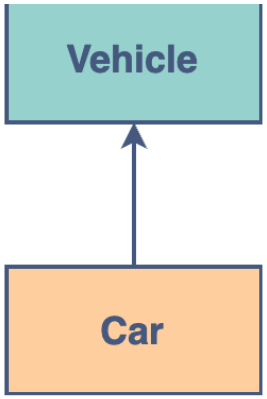
IS A Relationship

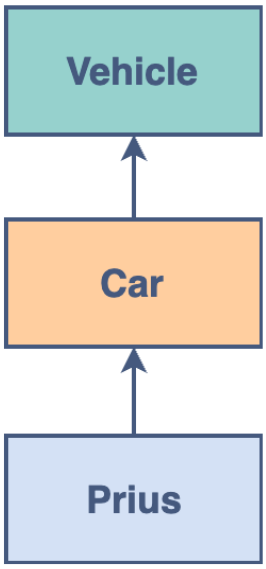
Inheritance provides a way to create a new class from an existing class. The new class is a specialized version of the existing class such that it inherits all the *non-private* fields (*variables*) and *methods* of the existing class. The existing class is used as a starting point or as a *base* to create the new class.

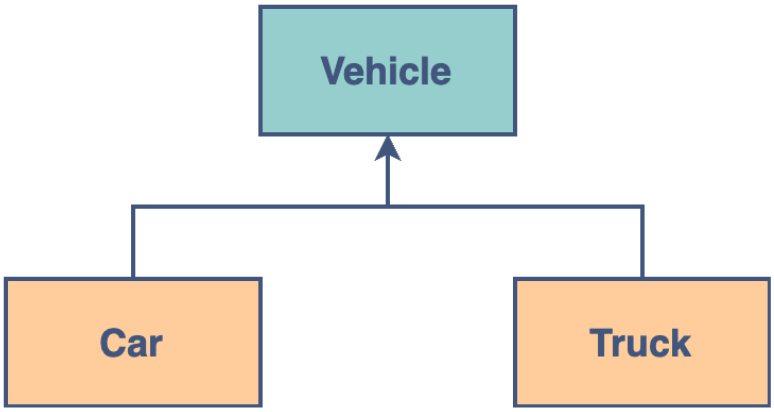


Types of Inheritance

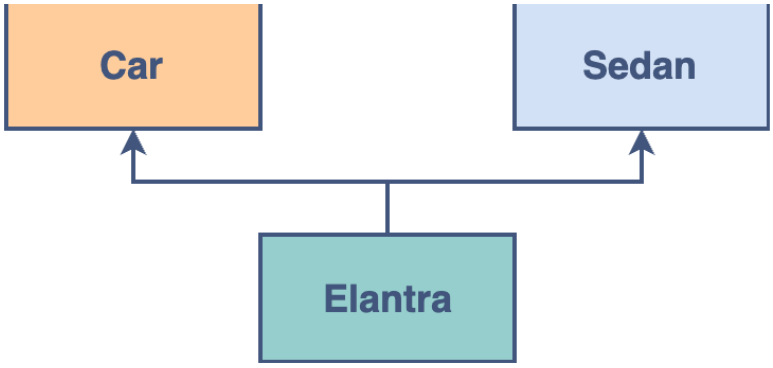
- **Single Inheritance**



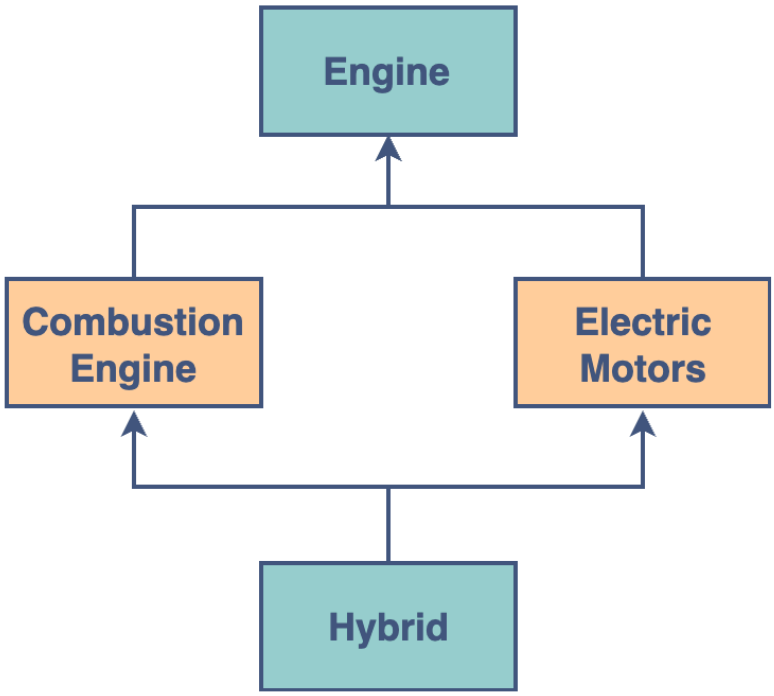
- Multi-level Inheritance
  - 

- Hierarchical Inheritance
  - 

- Multiple Inheritance
  -



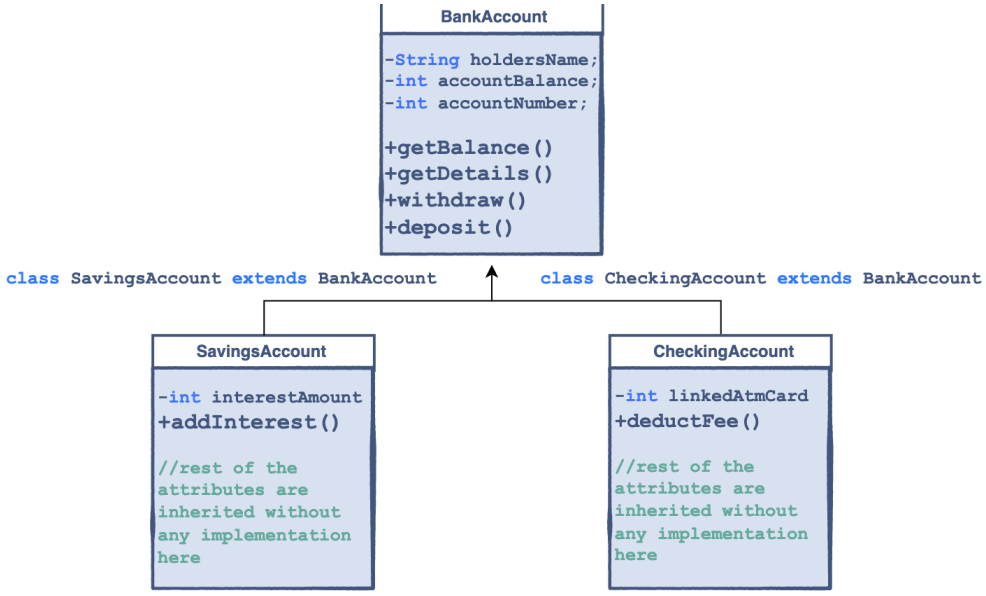
- Hybrid Inheritance
  -



Note: In Java, **Multiple** and **Hybrid** inheritance are applicable using **interface** only.

Advantages of Inheritance

- **Re-usability/Avoiding Duplication** of Code
  -



- **Extensibility**
  - add a new method to super class, then all sub-classes have this new method

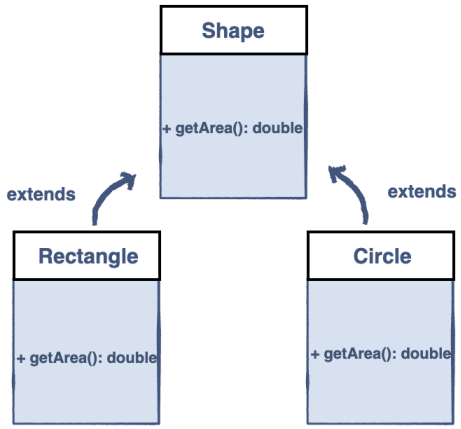
Polymorphism(多态)

- **Static** Polymorphism - **Overload** (same class) - **compile time**
- **Dynamic** Polymorphism - **Override** (child class) - **run time**

In programming, ***polymorphism*** refers to the same object exhibiting different **forms and behaviors**.

For example, take the Shape Class. The exact shape you choose can be anything. It can be a rectangle, a circle, a polygon or a diamond. So, these are all shapes but their properties are different. This is called **Polymorphism**.

Here we consider the example of a **Shape** class, which is the base class while many shapes like *Rectangle and Circle* extending from the base class are derived classes. These classes contain the **getArea()** method which calculates the area for the respective shape.



Code: oop

Method Overriding

Method Overriding

> *Method overriding* is the process of redefining a parent class's method in a subclass.

Advantages of the method overriding

Method overriding is very useful in OOP. Some of its **advantages** are stated below:

- The derived classes can give **their own specific implementations** to inherited methods without modifying the parent class methods.
- For any method, a child class can use the implementation in the parent class or make its own implementation.

Question: Can we overide final class and final method ?

Why **overriding** is **runtime** ? Answer

```
List<Integer> lst = new ArrayList<>();
```

Override vs. Overloading

Method Overloading	Method Overriding
Overloading happens at **compile time.**	Overriding happens at **runtime**
Gives **better performance** because the binding is being done at compile time.	Gives **less performance** because the binding is being done at run time.
Private and **final** methods can be **overloaded**.	**Private** and **final** methods can **NOT** be overridden.
Return type of method does not matter in case of method overloading.	Return type of method must be the same in the case of overriding.
Arguments must be different in the case of overloading.	Arguments must be the same in the case of overriding.
It is being done in the **same class**.	Base and **derived(child) classes** are required here.
Mostly used to increase the **readability** of the code.	Mostly used to provide the **implementation** of the method that is already provided by its base class.

Overloading: diff num of **arguments**, diff type **aruments**, **same method name**

Question: Same arguments, same method name, diff return type. is it overloading? is it allowed in java?

```
public int add(int a, int b) {
```

```
public int add(int a, int b) {
    return a + b;
}

public String add(int a, int b) {
    return "a + b";
}
```

Question: what is **static** polymorphism? what is **dynamic** polymorphism

Abstract Classes and Methods

Keyword: **abstract**

Question: what is the relationship between **OOP** and **abstract class / interface**?

Abstract Methods

```
public abstract void makeSound(String s);

//dog.makeSound() --> bark

//cat.makeSound() --> meow

//human.makeSound() --> speak


//normal method
public void makeSound(String s){
  //make sound
  System.out.println(s);
}
```

Rule to be followed:

- **No Method body** `{...}`
- can declared inside an **abstract class** and **interface**
  - **NON-abstract classes cannot have abstract methods**
- **can NOT** be declared **private**
  - it has to be implemented in some other class

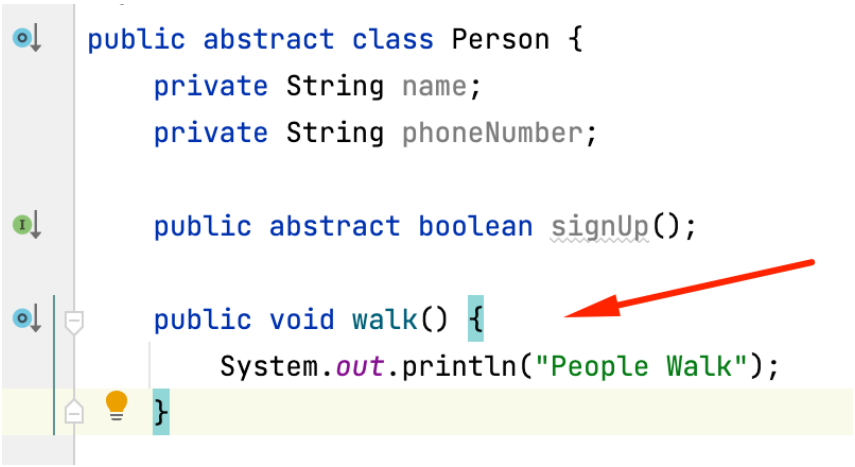
继承时候会被强制写Override该方法

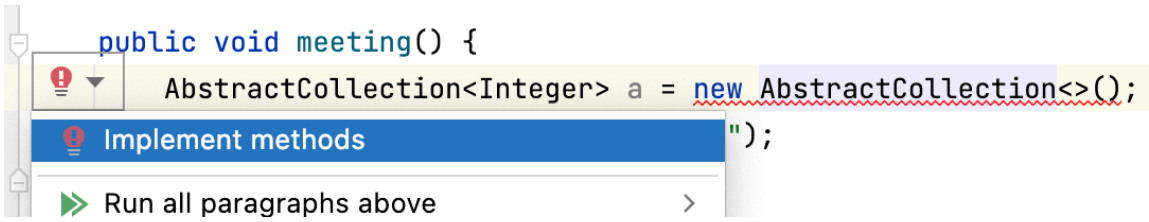
Abstract Class

```
abstract class ClassName {
  // code
}
```

```
public abstract class AbstractCollection<E> implements Collection<E> {
    Sole constructor. (For invocation by subclass constructors, typically implicit.)
    protected AbstractCollection() {
    }
```

Rule to be followed:

- An **abstract class** can have everything else as same as a normal Java class has **i.e.** `constructor, static variables and methods`.
  - **Non-abstract/Normal methods** can be implemented in an **abstract class** (At least in Java 8)
  - 

- An abstract class ***cannot*** be **instantiated**
  - You can **NOT** do `Person person = new Person()`
- Child classes **must implement all the abstract methods** declared in the parent abstract class.
  - 

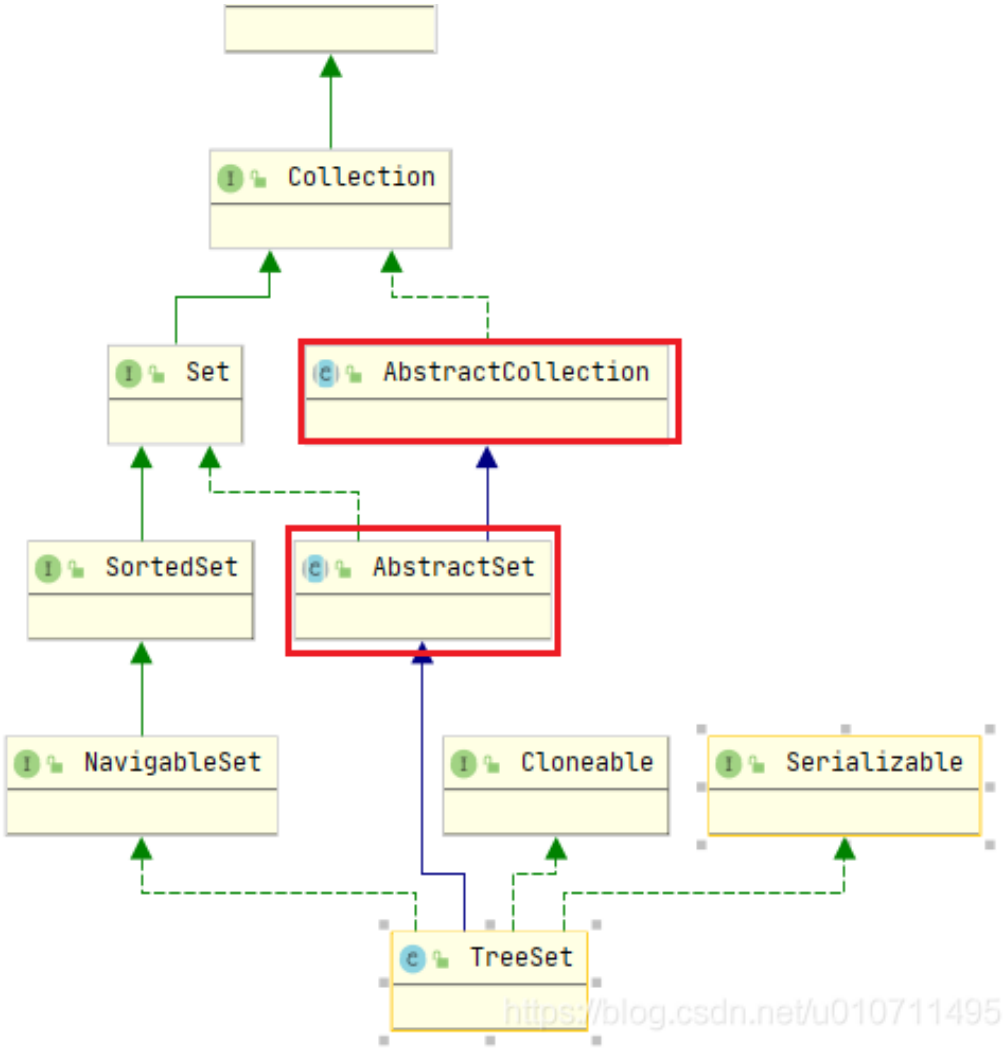
  -

```
AbstractCollection<Integer> a = new AbstractCollection() {
```

```
    /**
     * Returns an iterator over the elements contained in this collection.
     *
     * @return an iterator over the elements contained in this collection
     */
    @Override
    public Iterator iterator() {
        return null;
    }

    @Override
    public int size() {
        return 0;
    }
};
```

- Iterable



Interface

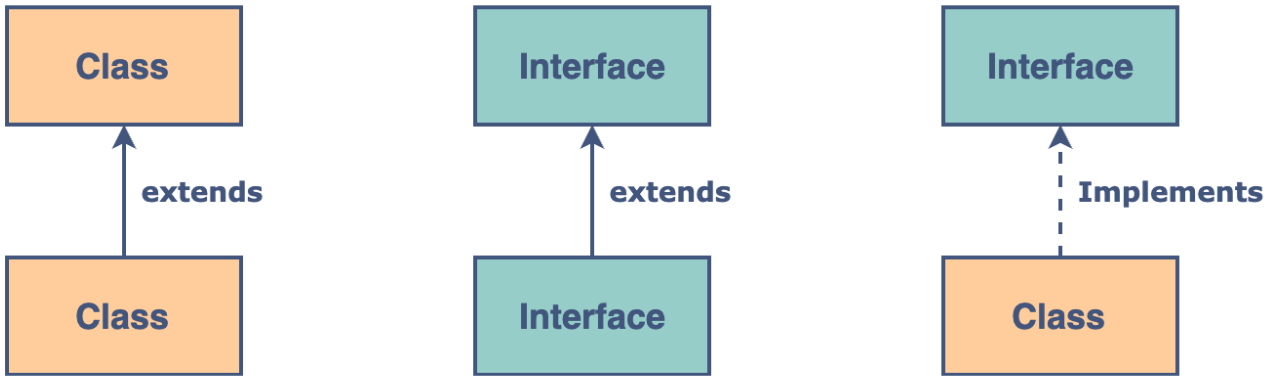
Question: what is the keyword of **Interface**?

Question: list some Interfaces in Java.

Rules to follow:

- An **interface** can have:
  - `abstract` method(s)
  - `default` method(s) - Java 8 new features
  - `static` method(s)
  - `private` method(s)
  - `private static` method(s)
  - `public static final` variable(s)
- methods -> by **default** `public`
- varibles -> by **default public static final**

    - 我们写代码时候还需要写public么？
- **cannot** be **instantiated**.
- An **interface** cannot be declared **private** or **protected**.
- To use an interface, a class **must** `implement` all of the `abstract` method(s) declared in it.
- **cannot** have **constructor(s)** in it
    - how about abstract class for the above two items ?
    - Why abstract class have construtors? (cause non-static fields)
- An interface can extend from another interface
    -



> **Note:** A class uses the keyword `implements` to use an interface but an interface uses the keyword `extends` to use *another* interface.

```java
public interface Vehicle {
    void cleanVehicle();
    default void startVehicle() {
        System.out.println("Vehicle is starting");
    }
}
```

Important Questions

Question: Why Java add default method in Java 8 ? Video

Question: Interface 可以多继承，那么两个interface的同样的method的情况下该怎么办？code

Question: Static Methods in interfaces ? code

Functional Interface - java 8

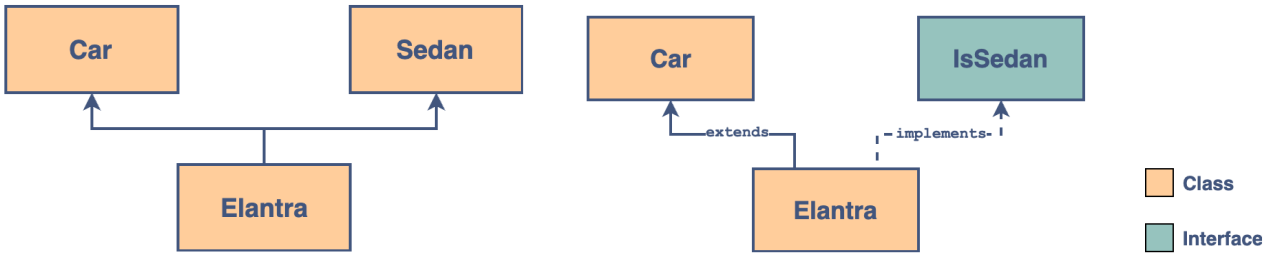
> An interface that has **a single abstract method** is called a **functional interface**.
>
> The functional interface is used by **lambda** expressions

Question: what does **a single abstract method**?

```java
@FunctionalInterface
public interface Functional {
    void doSomething(a);

    default void foo() {
        System.out.println("foo");
    }
}
```

Multiple Inheritance




```java
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable
{
  // code
}
```

Desgin Pattern (OOD)

Aggregation and Composition

Question: Guess Why I introduce **Aggregation** and **Composition** here?

Question: Design Pattern/SOLID 围绕的核心是什么？

Relationships Between Classes

- Is a - **Inheritance**

- Part-of : **Composition**(e.g. 4 Wheels --> Car)
- Has-a : **Aggregation**(e.g. Car --> 4 wheels)

Aggregation

> In **aggregation**, the lifetime of the owned object does not depend on the lifetime of the owner.

```
    public Person(String n, Country c) {
      name = n;
      country = c;
    }

Country country = new Country("Utopia", 1);
Person user = new Person("Darth Vader", country);
```

```
class Country {
    private String name;
    private int population;

    public Country(String n, int p) {
      name = n;
      population = p;
    }
    public String getName() {
      return name;
    }
}

class Person {
    private String name;
    private Country country; // An instance of Country class

    public Person(String n, Country c) {
      name = n;
      country = c;
    }

    public void printDetails() {
      System.out.println("Name: " + name);
      System.out.println("Country: " + country.getName());
    }
}

```

```
class Main {
  public static void main(String args[]) {
    Country country = new Country("Utopia", 1);
    {
      Person user = new Person("Darth Vader", country);
      user.printDetails();
    }
    // The user object's lifetime is over

    System.out.println(country.getName()); // The country object still exists!
  }
}
```

Composition

> In **composition**, the lifetime of the owned object depends on the lifetime of the owner.

```
  public Car(String col, int cap, int nt, int nod) {
    this.eObj = new Engine(cap);
    this.tObj = new Tires(nt);;
    this.dObj = new Doors(nod);

    color = col;
  }

Car cObj = new Car("Black", 1600, 4, 4);
```

```
class Engine {
  private int capacity;

  public Engine(){
    capacity = 0;
  }

  public Engine(int cap) {
    capacity = cap;
  }

  public void engineDetails() {
    System.out.println("Engine details: " + capacity);
  }
}

```

```
class Tires {
  private int noOfTires;

  public Tires() {
    noOfTires = 0;
  }

  public Tires(int nt) {
    noOfTires = nt;
  }

  public void tireDetails() {
    System.out.println("Number of tyres: " +  noOfTires);
  }

}

class Doors {

  private int noOfDoors;

  public Doors() {
    noOfDoors = 0;
  }

  public Doors(int nod) {
    noOfDoors = nod;
  }

  public void doorDetails() {
    System.out.println("Number of Doors: " + noOfDoors);
  }

}

class Car {

  private Engine eObj;
  private Tires tObj;
  private Doors dObj;
  private String color;

  public Car(String col, int cap, int nt, int nod) {
    this.eObj = new Engine(cap);;
    this.tObj = new Tires(nt);;
    this.dObj = new Doors(nod);
```

```
63
64      color = col;
65    }
66
67    public void carDetail() {
68      eObj.engineDetails();
69      tObj.tireDetails();
70      dObj.doorDetails();
71      System.out.println("Car color: " + color);
72    }
73
74  }
75
76  class Main {
77
78    public static void main(String[] args) {
79      Car cObj = new Car("Black", 1600, 4, 4);
80      cObj.carDetail();
81    }
82  }
```

Answer:

Question: Guess Why I introduce **Aggregation** and **Composition** here? (OOP is not lose coupling) (decouple的工具)

Question: Design Pattern/SOLID 围绕的核心是什么? (Decouple)

Design Pattern - Singleton(单例)

Steps:

1. static volatile **variable**
2. make **constructor** be **private**
3. **static** synchronized **getInstance** method
4. make sure **thread safe**

Eager Load

```
public class Singleton {
    // 1, private static variable
    private static Singleton instance = new Singleton();

    // 2, make constructor be private
    // 保证不能new， 一旦可以new，就可以建造很多instance， 则就不再是singleton。
    private Singleton() {
    }

    // 3. static getInstance method
    // static保证在没有instance的情况下，可以调该方法。
    public static Singleton getInstance() {
        return instance;
    }
}

Singleton is NOT null
Singleton.getInstance(); //faster
```

Lazy Load

In this implementation, the Singleton instance is created when the `SingletonHolder` class is loaded and initialized. This occurs only when the `getInstance()` method is called for the first time, ensuring lazy initialization. Since class loading and initialization are thread-safe in Java, this approach guarantees that the Singleton instance is created only once, even in a multi-threaded environment.

```
public class Singleton {

    // Private constructor to prevent instantiation from outside the class
    private Singleton() {
    }

    // Static inner class to hold the Singleton instance
    private static class SingletonHolder {
        private static final Singleton INSTANCE = new Singleton();
    }

    // Public static method to get the Singleton instance
    public static Singleton getInstance() {
        return SingletonHolder.INSTANCE;
    }
}

```

Java 中的单例模式（Singleton）是一种设计模式，其目标是确保一个类只有一个实例，并提供全局访问点。静态内部

类的使用有几个好处：

1. **延迟加载**：静态内部类的特性保证了只有在被调用时才会被加载，实现了延迟加载。
2. **线程安全**：Java 类加载机制保证了类只会被加载一次，这种方式也自然实现了线程安全。

`SingletonHolder` 是一个静态内部类，它包含了一个 `Singleton` 类型的静态常量 `INSTANCE`。只有当 `getInstance()` 方法被调用时，`SingletonHolder` 才会被加载，`INSTANCE` 才会被初始化。而且，由于 `INSTANCE` 是 `final` 的，所以一旦 `INSTANCE` 被初始化，就不可能被改变。

由于类加载是线程安全的，所以这种方式能保证在多线程环境下，无论调用多少次 `getInstance()` 方法，都只会得到同一个 `Singleton` 实例。