

Java & OOP

Repo

JAVA Basic

JDK vs. JRE vs. JVM

Variable Types

Primitive types(原始类型数据, 能用 “==” 比较)

Reference types(引用类型数据)

String Constant Pool(字符串常量池)

Final Keyword

Variable

Method

Class

Static

Static Variable and Static Block

Static Method

Static Class

JVM Load

Java Comment

Java commands

OOP*

Class and Object

Components

Constructor

Encapsulation(封装)

Access Modifiers

Inheritance(继承)

Types of Inheritance

Advantages of Inheritance

Polymorphism(多态)

Method Overriding

Advantages of the method overriding

Override vs. Overloading

Abstract Classes and Methods

Abstract Methods

Abstract Class

Interface

Important Questions

Functional Interface - java 8

Multiple Inheritance

Abstract class vs. Interface

Design Pattern (OOD)

Aggregation and Composition

Aggregation

Composition

Design Pattern - Singleton(单例)

Eager Load

Lazy Load

Java & OOP

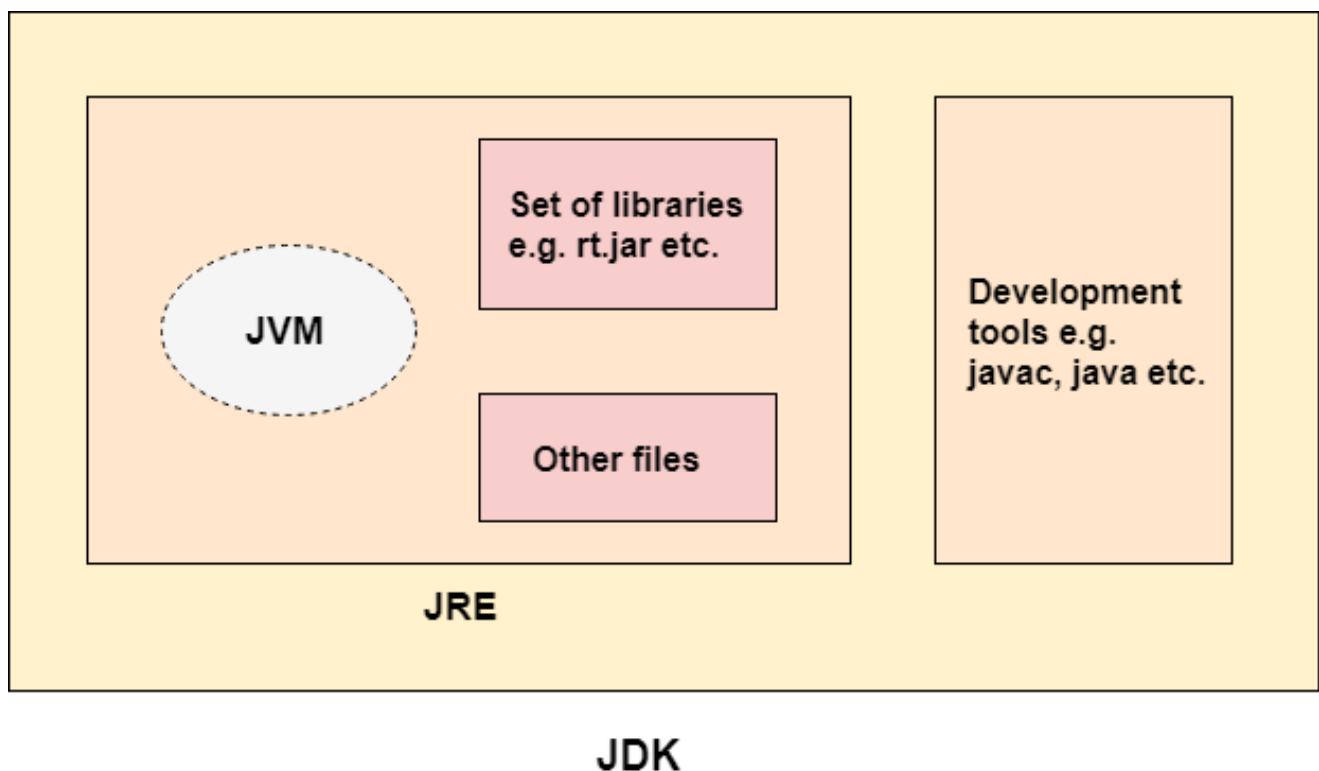
Repo

<https://github.com/TAlRich/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.

```
1  int myNum = 5;           // Integer (whole number)
2  float myFloatNum = 5.99f; // Floating point number
3  char myLetter = 'D';     // Character
4  boolean myBool = true;   // Boolean
5  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

```
1  public int add(int a, int b){
2      return a + b;
3  }
4
```

```

5 public int changeAge(User u1, User u2) {
6     u1.setAge(18);
7     return u1.age + u2.age;
8 }
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 }

```

Pass by value: makes a copy in memory of the parameter's value, or a copy of the contents of the parameter.

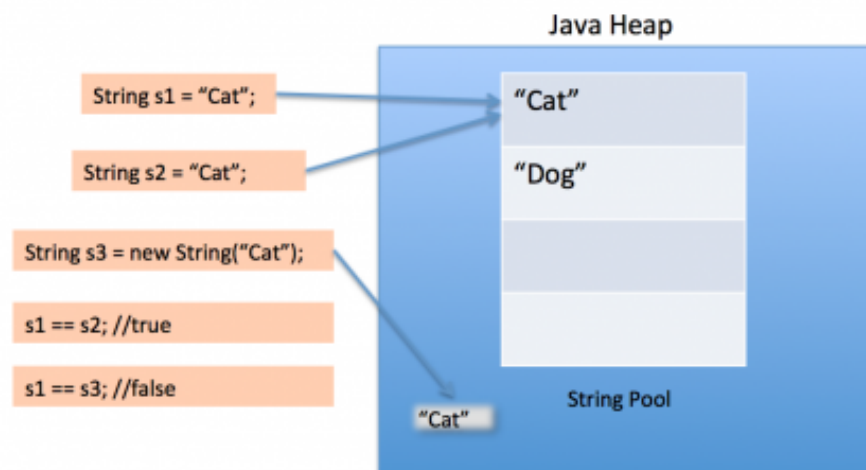
Pass by reference: a copy of the address (or reference) to the parameter is stored rather than the value itself.

Questions: is java passing by value or passing by Ref? [Answer](#)

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 ?

(security, thread safety, performance optimization, and predictable behavior.)

Final Keyword

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**



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;  
    }  
}
```

```
13 public class EmployeeData {  
14  
15     @Override  
16     public static List<Employee> getEmployees() {  
17         List<Employee> list = new ArrayList<>();  
18  
19         list.add(new Employee( id: 1001, name: "Yun Ma", age: 50, salary: 30000.00));  
20         list.add(new Employee( id: 1002, name: "Huateng Ma", age: 49, salary: 22222.22));  
21         list.add(new Employee( id: 1004, name: "Jun Lei", age: 43, salary: 12234.12));  
22         list.add(new Employee( id: 1005, name: "Bill Gates", age: 65, salary: 999999));  
23         list.add(new Employee( id: 1003, name: "Yanhong Li", age: 30, salary: 123123));  
24         list.add(new Employee( id: 1007, name: "Zhengfei Ren", age: 78, salary: 66666));  
25         list.add(new Employee( id: 1006, name: "Mark Elliot Zuckerberg", age: 29, salary: 88888));  
26  
27         return list;  
28     }  
}
```

Questions:

1. Can static method access non-static variables?
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;

/**
 * @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();
    }
}

/**
 * @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)

```
13 public class JvmLoad {
14     public static void main(String[] args) {
15         Demo demo = new Demo();
16     }
17 }
18
19 class Demo {
20     // default value is 0
21     private static int n1;
22     private static final String s1 = "static variable";
23     private static Set<Employee> employees = new HashSet<>();
24     private String s2 = "non static variable";
25
26     static {
27         System.out.println("***** 1, static block is called *****");
28         System.out.println("***** 2, check values of static variables *****");
29         System.out.println(n1);
30         System.out.println(s1);
31         System.out.println(employees);
32         System.out.println("**** done ****\n");
33         // static method: EmployeeData.getEmployees()
34         employees.addAll(EmployeeData.getEmployees());
35
36         // in static block, call non-static variable
37         System.out.println(s2);
38     }
39
40     public Demo() {
41         System.out.println("***** 3, Constructor is called *****");
42         System.out.println(n1);
43         System.out.println(s1);
44         employees.forEach(System.out::println);
45         System.out.println("---- s2 have value now ----");
46         System.out.println(s2);
47         System.out.println("**** done *****");
48     }
49 }
```

***** 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());  
    }  
}
```

Variables

Evaluate expression (↵) or add a watch (⌘⌘↵)

- static members of Demo
 - n1 = 0
 - s1 = "static variable"
 - employees = {HashSet@656} size = 0
 - n1 = 0



Java 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

```
1 $> javac HelloWorld.java // HelloWorld.class
2 $> java HelloWorld
3 $> javadoc -d HelloWorld HelloWorld.java
```

OOP*

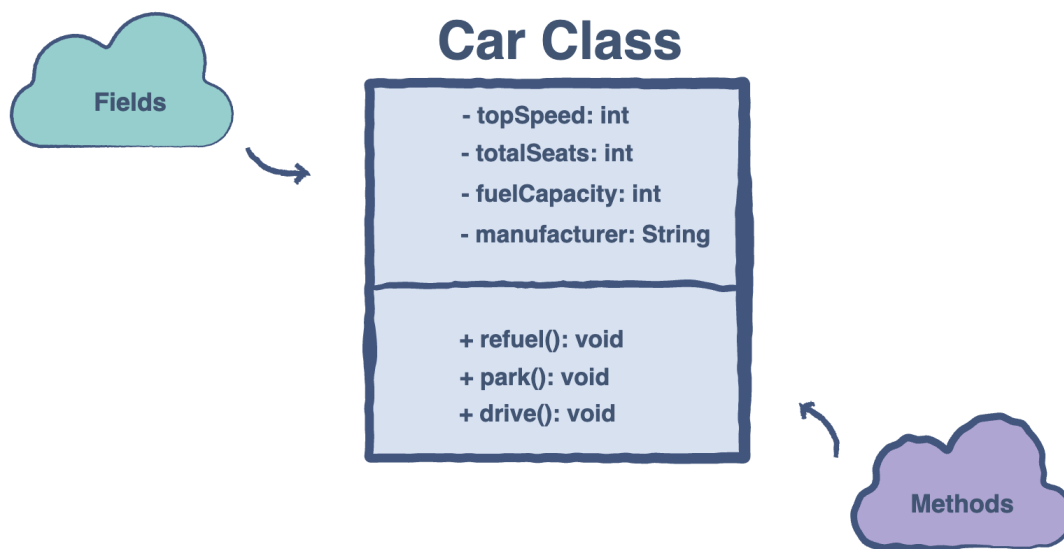
Class and Object

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/Properties
- 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(Body carbody, wheel[] wheels, Steer steer...)`
- `public Car(Sting brand/maker, String carType, String color, int seats){...}`
- `public Car(Sting brand/maker, String carType, String color, int seats, String autopilot){...}`
- 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

Questions: 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).

```
/* 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;
    }
}
```

```
/* 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)

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.

Square



**IS A
Shape**

Java



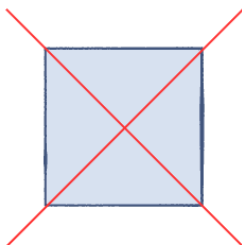
**IS A
Programming Language**

Car



**IS A
Vehicle**

Square



**IS A
Corners**

Java



**IS A
Syntax**

Car



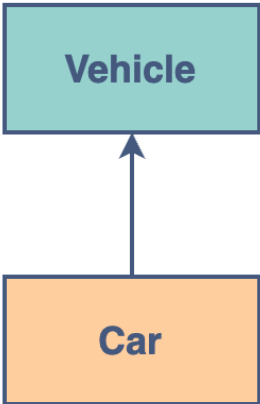
**IS A
Steering**

Types of Inheritance

types of inheritance

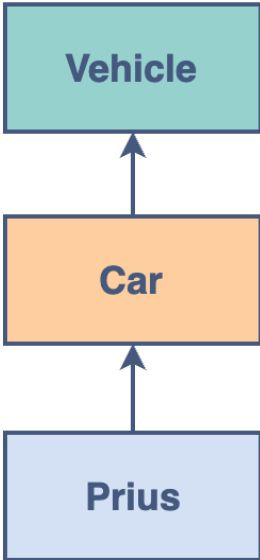
- Single Inheritance

-



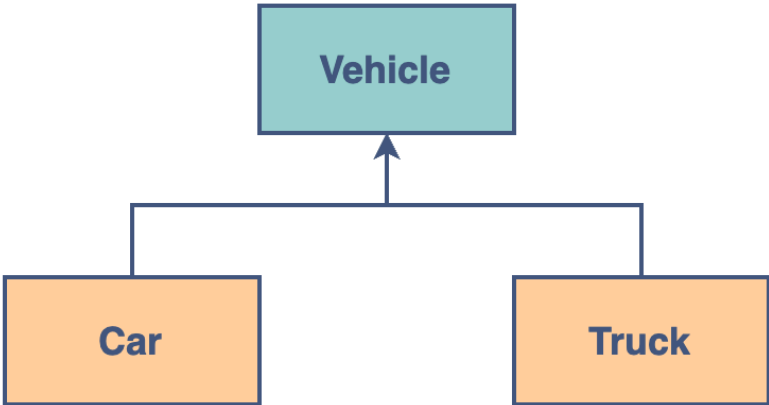
- Multi-level Inheritance

-



- Hierarchical Inheritance

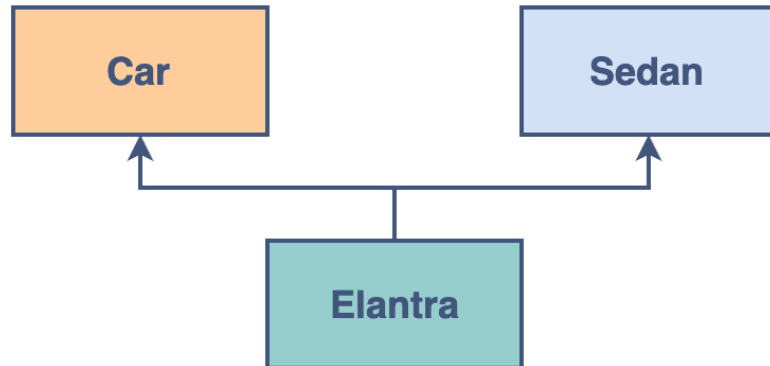
-



- Multiple Inheritance

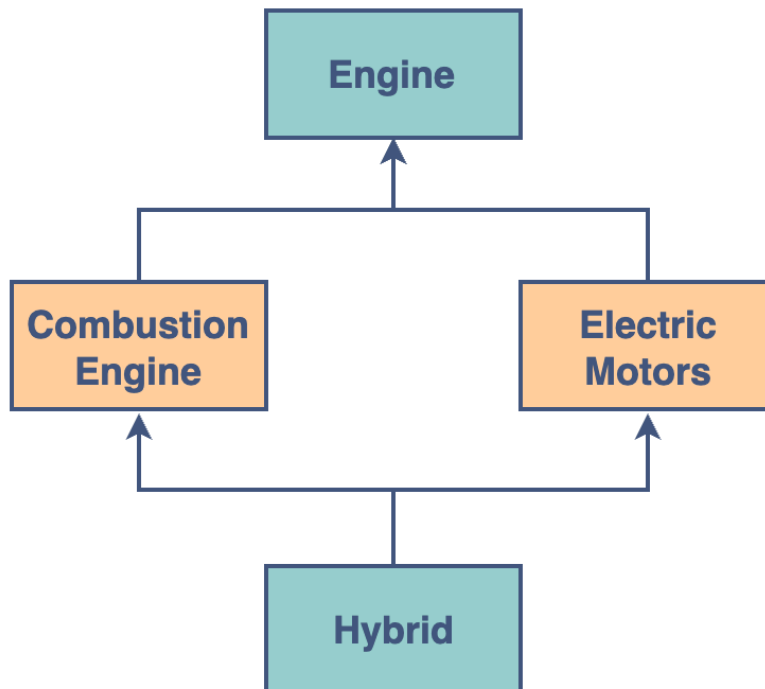
- Multiple Inheritance

- o



- Hybrid Inheritance

- o

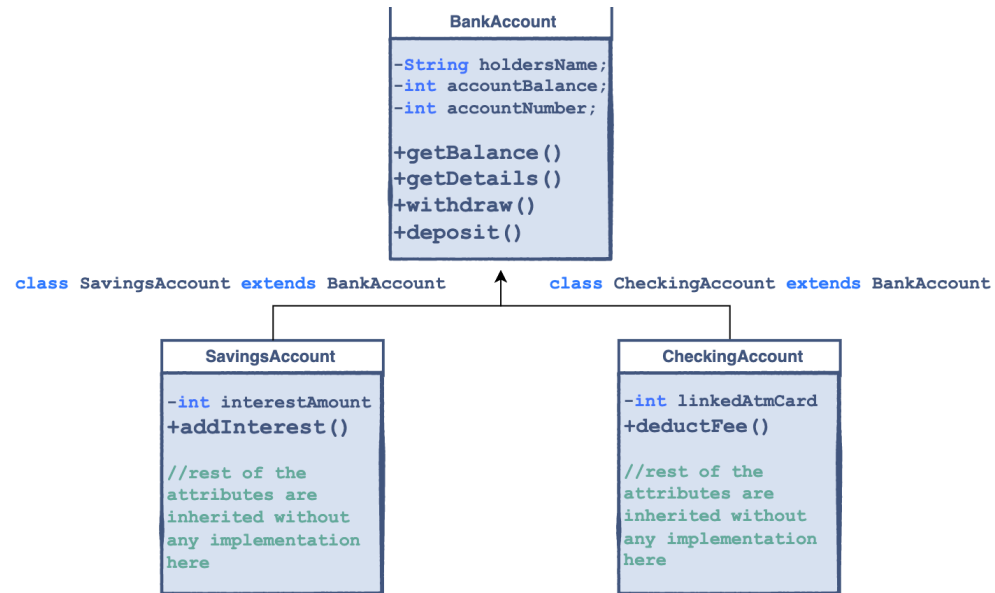


Note: In Java, **Multiple** and **Hybrid** inheritance are applicable using **interface** only.

Advantages of Inheritance

- Re-usability/Avoiding Duplication of Code

- o



- **Extensibility**

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

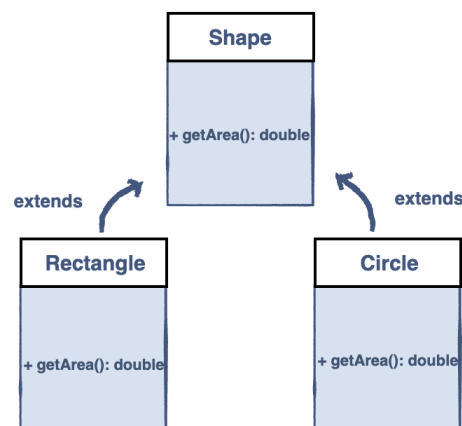
Polymorphism(多态)

- **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 override final class and final method ?

Why **overriding** is **runtime** ? [Answer](#)

```
1 | 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 **arguments**, **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

```

1  abstract class Animal{
2      //common behavior
3      abstract void makeSound(String s);
4  }
5
6  class Dog extends Animal{
7      @Override
8      public void makeSound(String s){
9          System.out.println("bak~");
10     }
11 }
12
13 //cat.makeSound() --> meow
14 class Cat extends Animal{
15     @Override
16     public void makeSound(String s){
17         System.out.println("meow~");
18     }
19 }
20
21 //human.makeSound() --> speak
22
23
24 //normal method
25 public void makeSound(String s){
26     //make sound

```



```
27     System.out.println(s);
28 }
```

Rule to be followed:

- **No Method body** {...}
- can declared inside an **abstract class** and **interface**
 - **non-abstract classes cannot have abstract methods**
- **cannot** 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

```
1  abstract class ClassName {
2      // code
3  }
```

```
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)
 -

```
public abstract class Person {
```


```

public abstract class Person {
    private String name;
    private String phoneNumber;

    public abstract boolean signUp();

    public void walk() {
        System.out.println("People Walk");
    }
}

```



- 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.

```

public void meeting() {
    AbstractCollection<Integer> a = new AbstractCollection<>();
}

```

Implement methods

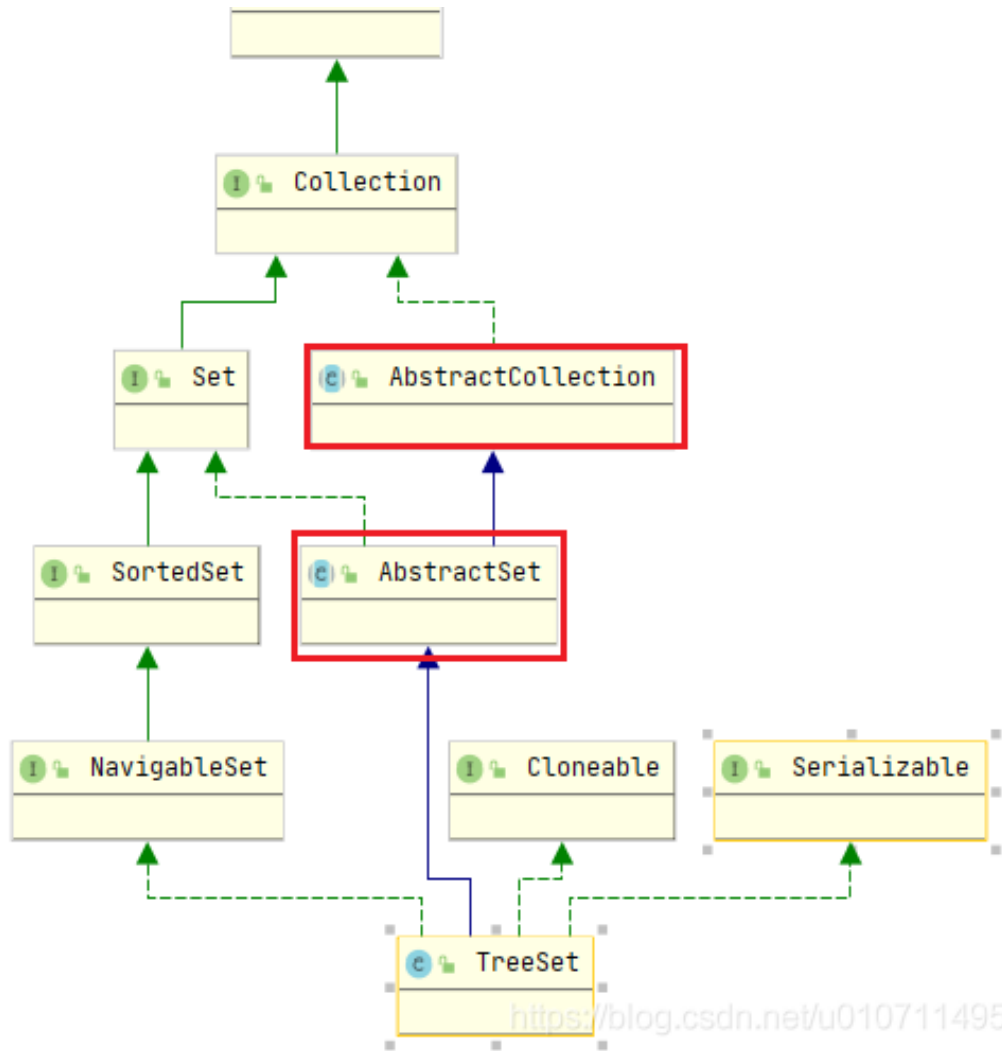
Run all paragraphs above

```

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;
    }
};

```



Interface

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

Question: list some Interfaces in Java.

sumOf(int x, int y); 5 + 5

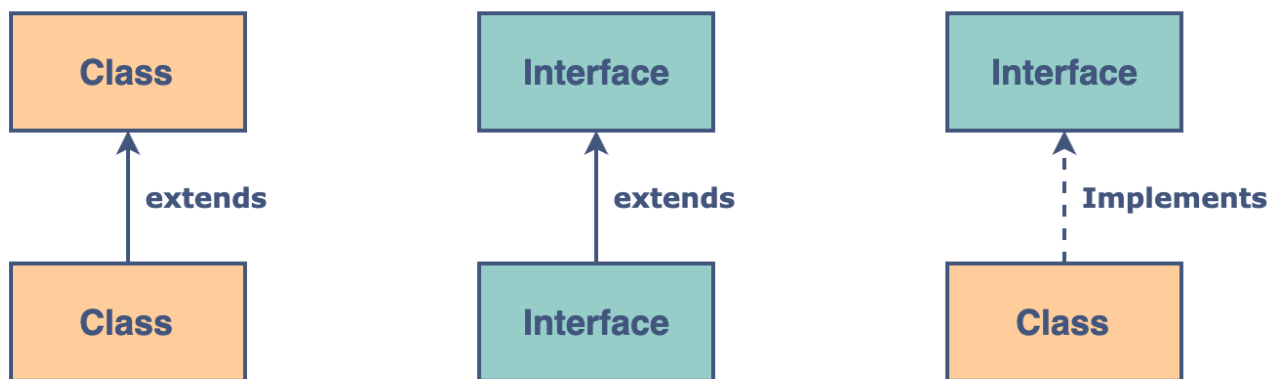
5 + 5 ==> 10

1+1..+1 1+1..+1 =>10

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`
- variables -> by **default** `public static final`
- **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 ?
 - Why abstract class have constructors ?
- 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.

```

1 public interface Vehicle {
2     void cleanVehicle();
3
4     default void startVehicle() {
5         System.out.println("Vehicle is starting");
6     }
7 }
  
```

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? codedemo

Question: Static Methods in interfaces ? codedemo

Functional Interface - java 8

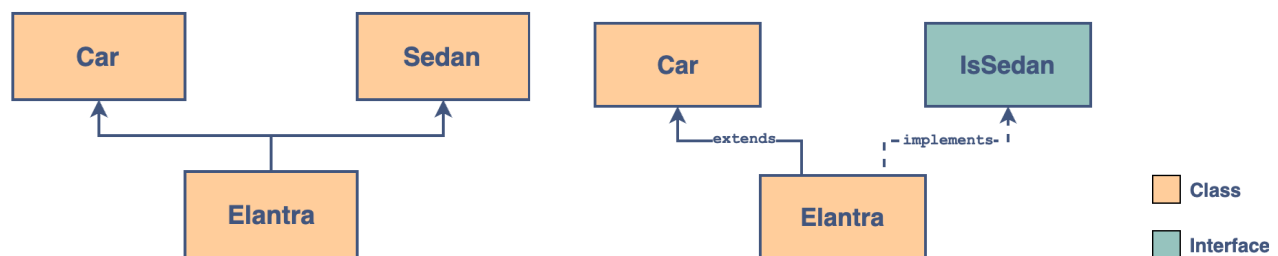
An interface that has **a single abstract method** is called a **functional interface**.

The functional interface is used by **lambda** expressions

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

```
1 @FunctionalInterface
2 public interface Functional {
3     void doSomething(a); //abstract method ONLY ONE
4
5     default void foo() {
6         System.out.println("foo");
7     }
8 }
```

Multiple Inheritance



```
1 public class LinkedList<E>
2     extends AbstractSequentialList<E>
3     implements List<E>, Deque<E>, Cloneable, java.io.Serializable
4 {
5     // code
6 }
```

Abstract class vs. Interface

	Interface	Abstract class
Usage	Implements	extends
Multiple Inheritances	Implement several Interfaces	Only one abstract class
When to use	Future enhancement	To avoid independence
Access Modifiers	The interface does not have access modifiers. Everything defined inside the interface is assumed public modifier .	Abstract Class can have an access modifier .
When to use	It is better to use interface when various implementations share only method signature . Polymorphic hierarchy of value types.	It should be used when various implementations of the same kind share a common behavior .
Implementation	Interface can have only abstract methods . An abstract class can have abstract and non-abstract methods. From Java 8 , it can have default and static methods also. From Java 9 , it can have private concrete methods as well.	An abstract class can give complete, default code which should be overridden
Usage	Interfaces help to define the peripheral abilities of a class.	An abstract class defines the identity of a class.
Defined fields	only static and final variables.	Abstract class can have final, non-final, static and non-static variables.
Constructor or destructors	An interface cannot declare constructors or destructors.	An abstract class can declare constructors and destructors.
Abstract keyword	In an abstract interface keyword, is optional for declaring a method as an abstract.	In an abstract class, the abstract keyword is compulsory for declaring a method as an abstract.

Important Reasons For Using Interfaces

- Interfaces are used to **achieve abstraction**.
- Designed to support dynamic method resolution at run time
- It helps you to achieve **loose coupling**.
- Allows you to separate the definition of a method from the inheritance hierarchy

Important Reasons For Using Abstract Class

- Abstract classes offer **default functionality** for the subclasses.
- Provides a **template** for future specific classes
- Helps you to define a **common interface** for its subclasses
- Abstract class allows code **reusability**.

[Further Reading](#)

Design 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, two arms are **part of** the woman/man)
- Has-a : **Aggregation**(e.g. Car --> 4 wheels, A man/woman **has** two arms)

Aggregation

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

```
1 public Person(String n, Country c) {
2     name = n;
3     country = c;
4 }
5
6 Country country = new Country("Utopia", 1);
7 Person user = new Person("Darth Vader", country);
```

```
1 class Country {
2     private String name;
3     private int population;
4
5     public Country(String n, int p) {
6
7         name = n;
```

```

7     population = p;
8 }
9 public String getName() {
10     return name;
11 }
12 }
13
14 class Person {
15     private String name;
16     private Country country; // An instance of Country class
17
18     public Person(String n, Country c) {
19         name = n;
20         country = c;
21     }
22
23     public void printDetails() {
24         System.out.println("Name: " + name);
25         System.out.println("Country: " + country.getName());
26     }
27 }
28
29 class Main {
30     public static void main(String args[]) {
31         Country country = new Country("Utopia", 1);
32         {
33             Person user = new Person("Darth Vader", country);
34             user.printDetails();
35         }
36         // The user object's lifetime is over
37
38         System.out.println(country.getName()); // The country object still exists!
39     }
40 }

```

Composition

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

```

1 public Car(String col, int cap, int nt, int nod) {

```



```
2     this.eObj = new Engine(cap);
3     this.tObj = new Tires(nt);
4     this.dObj = new Doors(nod);
5
6     color = col;
7 }
8
9 Car cObj = new Car("Black", 1600, 4, 4);
```

```
1 class Engine {
2     private int capacity;
3
4     public Engine(){
5         capacity = 0;
6     }
7
8     public Engine(int cap) {
9         capacity = cap;
10    }
11
12    public void engineDetails() {
13        System.out.println("Engine details: " + capacity);
14    }
15 }
16
17 class Tires {
18     private int noOfTires;
19
20     public Tires() {
21         noOfTires = 0;
22     }
23
24     public Tires(int nt) {
25         noOfTires = nt;
26     }
27
28     public void tireDetails() {
29         System.out.println("Number of tyres: " + noOfTires);
30     }
31
32 }
33
34 class Doors {
```

```
35
36     private int noOfDoors;
37
38     public Doors() {
39         noOfDoors = 0;
40     }
41
42     public Doors(int nod) {
43         noOfDoors = nod;
44     }
45
46     public void doorDetails() {
47         System.out.println("Number of Doors: " + noOfDoors);
48     }
49
50 }
51
52 class Car {
53
54     private Engine eObj;
55     private Tires tObj;
56     private Doors dObj;
57     private String color;
58
59     public Car(String col, int cap, int nt, int nod) {
60         this.eObj = new Engine(cap);
61         this.tObj = new Tires(nt);
62         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
81         cObj.carDetail();
82     }
83 }
```

```
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** (optional)
2. make **constructor** be **private**
3. **static** synchronized **getInstance** method
4. make sure **thread safe**

Eager Load

```
1 public class Singleton {
2     // 1, private static variable
3     private static Singleton instance = new Singleton();
4
5     // 2, make constructor be private
6     // 保证不能new, 一旦可以new, 就可以建造很多instance, 则就不再是singleton。
7     private Singleton() {
8     }
9
10    // 3. static getInstance method
11    // static保证在没有instance的情况下, 可以调该方法。
12    public static Singleton getInstance() {
13        return instance;
14    }
15 }
16
17 Singleton is NOT null
18 Singleton.getInstance(); //faster
```

Lazy Load

Method #1

```
1 public class Singleton {
2     // 1, static volatile variable
3     private static volatile Singleton instance;
4
5     // 2, make constructor be private
6     // 保证不能new, 一旦可以new, 就可以建造很多instance, 则就不再是singleton。
7     private Singleton() {
8     }
9
10    // 3. static synchronized getInstance method
11    // static保证在没有instance的情况下, 可以调该方法。
12    public static Singleton getInstance() {
13        // 4, make sure thread safe, improve performance
14        if (instance == null) { // 可能是多个线程
15            // T1(first --> unlock), T2(wait)
16            synchronized (Singleton.class) { // 保证正能进入一个线程
17                //T1 is here first, and then comes T2
18                if (instance == null) {
19                    //T1 is here, T2...Tn will never be here.
20                    instance = new Singleton();
21                }
22            }
23        }
24
25        return instance;
26    }
27 }
28
29 Singleton is null!
30 Singleton.getInstance(); //a little slower, t1;
31 Singleton.getInstance(); //t2;
32
```

Method#2 (preferred way)

```
1 public class Singleton {
2     /**
3      * 静态内部类单例模式中实例由内部类创建, 由于 JVM 在加载外部类的过程中, 是不会加载静态
4      * 内部类的, 只有内部类的属性/方法被调用时才会被加载, 并初始化其静态属性。静态属性由于被
5      * static 修饰, 保证只被实例化一次, 并且严格保证实例化顺序。
6      */
7 }
```

```
7
8     // #1, make constructor be private
9     private Singleton() {
10    }
11
12    // #2, define a inner static class
13    private static class SingletonHolder {
14        private static final Singleton INSTANCE = new Singleton();
15    }
16
17    // #3, provide public access method
18    public static Singleton getInstance() {
19        return SingletonHolder.INSTANCE;
20    }
21 }
22
23 Singleton is null!
24 Singleton.getInstance();
25 Singleton.getInstance(); // the same instance
```