

复旦大学计算机科学技术学院



编程方法与技术

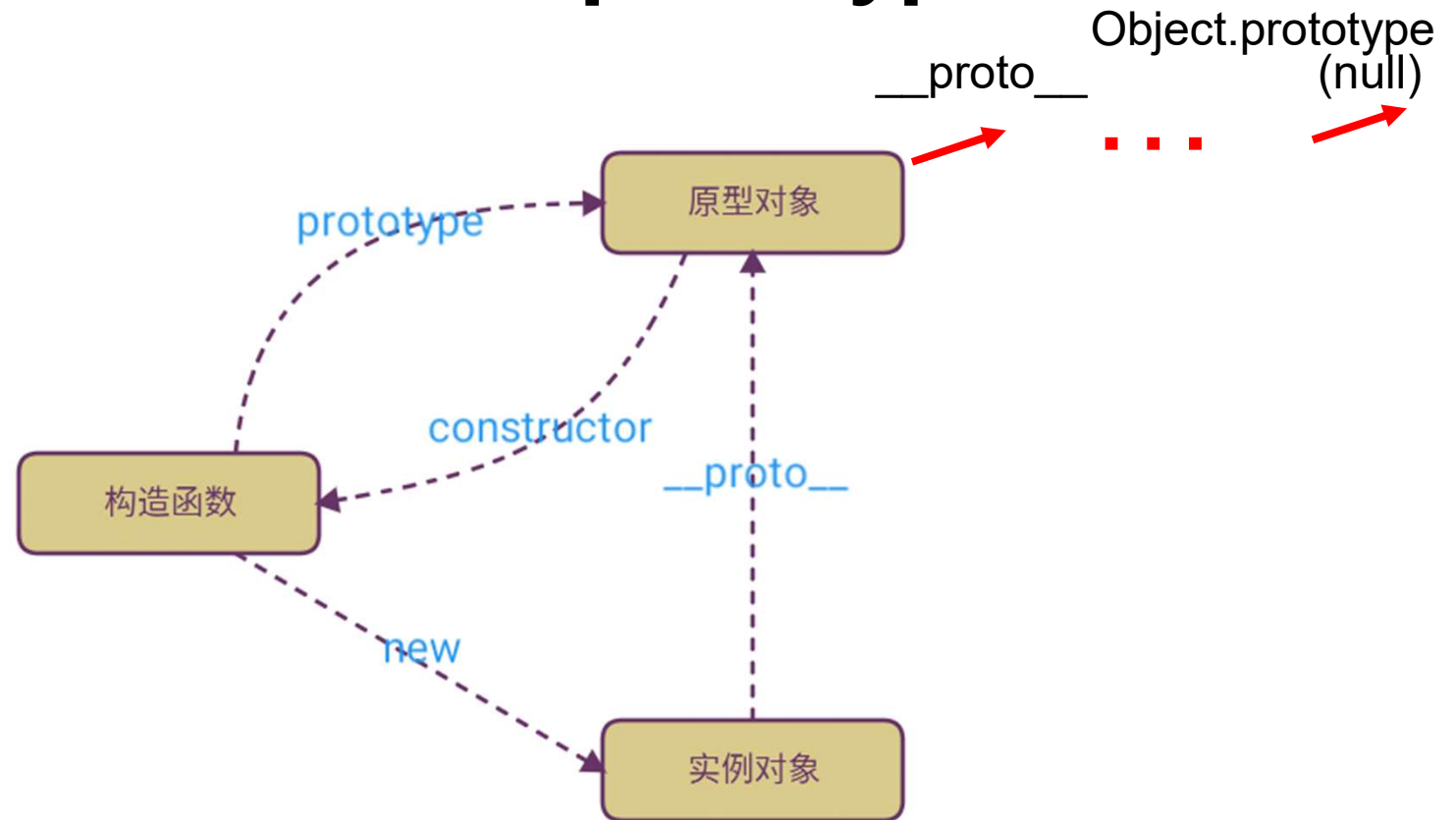
7.1. 面向对象复习

周扬帆

2021-2022第一学期

JS的对象继承方法

□ 方法1:通过构造函数的prototype实现继承



```
Dog.prototype = new Animal();  
var pp = new Dog();
```

JS的对象继承方法

□ 方法2: 通过构造函数实现继承

匿名

```
1 function Animal(name, sex) {  
2     this.name = name;  
3     this.sex = sex;  
4 }  
5  
6 Animal.prototype.getName = function () {  
7     return this.name;  
8 }  
9  
10 var cat = new Animal('white', 'male');  
11 cat.getName(); // white  
12  
13 function People(name, sex) {  
14     Animal.call(this, name, sex);  
15 }  
16  
17 People.prototype = new Animal('people', null);  
18  
19 var Chris = new People('Chris', 'male');  
20 Chris.getName(); // Chris
```

JS的对象继承方法

□ 方法3: 寄生组合继承

```
function derive(o) { //Object.create()
  function F() {
  }
  F.prototype = o;
  return new F();
}
```

```
var pp = new Dog('male');
```

```
console.log(pp1.gender);
console.log(pp1.getName());
```

```
function Animal(gender){
  this.gender = gender;
}
Animal.prototype.getName = function(){ return 'Animal';};
```

方法和数据剥离

```
function Dog(gender){
  Animal.call(this, gender);
  // ...
}
```

拷贝数据

```
var proto = derive(Animal.prototype);
proto.constructor = Dog;
Dog.prototype = proto;
```

“虚” 的原型，只负责连接方法

ES6 Class关键字

□ 语法糖

- 简化操作
- 易读

```
1  class Animal {
2      constructor(name, sex) {
3          this.name = name;
4          this.sex = sex;
5      }
6
7      getName() {
8          return this.name;
9      }
10 }
11
12 class People extends Animal {
13     constructor(name, sex) {
14         super(name, sex);
15     }
16 }
17
18 var Chris = new People('Chris', 'male');
19 Chris.getName(); // Chris
```

类的继承

□ 构造顺序

- 先父类，后子类

- 类内部的构造顺序

- static类的引用 = new ..., 先构造, 只做一次

- 类的引用 = new ..., 次之, 每次构造都做一次

- 如果声明时没有加 “= new ...”, 则仅声明, 不构造

- 再执行构造函数

判断题

- ❑ 类一定有构造方法会在创建该类的对象时被调用？
- ❑ 创建子类的对象时，先调子类自己的构造函数，然后调用父类的构造函数？
- ❑ 创建子类的对象时，一定有其父类的构造函数被调用？
- ❑ 创建子类的对象时，父类的构造函数只能通过子类在构造函数中用super关键字调用？
- ❑ 静态内部类会在外部类被构造时构造？
- ❑ 静态内部类的外部类会在内部类被构造时构造？

```

class AnotherClass {
    static {
        System.out.println("1");
    }
    AnotherClass() {
        System.out.println("2");
    }
}
public class Test {
    static {
        System.out.println("3");
    }
    AnotherClass test1 = new AnotherClass();
    static AnotherClass test2 = new AnotherClass();
    static AnotherClass test3 = new AnotherClass();
    static AnotherClass test4;
    Test() {
        System.out.println("4");
    }
    public static void main(String argv[]) {
        System.out.println("5");
    }
}

```

3
1
2
2
5


```

class People {
    String name;
    public People() {
        System.out.print(1);
    }
    public People(String name) {
        System.out.print(2);
        this.name = name;
    }
}

class Child extends People {
    People father;
    public Child(String name) {
        System.out.print(3);
        this.name = name;
        father = new People(name);
    }
    public Child() {
        System.out.print(4);
    }
}

People john = new Child("john");

```

132

多态Polymorphism

□ 引用的类型转换

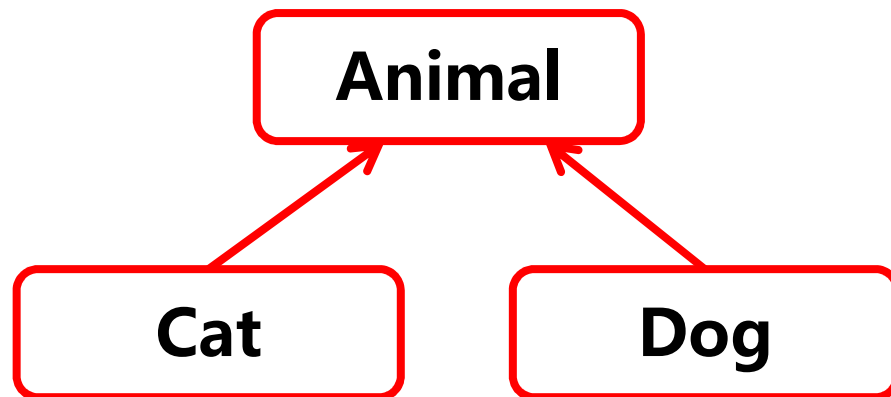
- 子类可以自动视为父类

 - 继承关系 = “是一种” 关系

- 父类变成子类需要显式的类型转换

- 除了继承关系，否则不允许类型转换

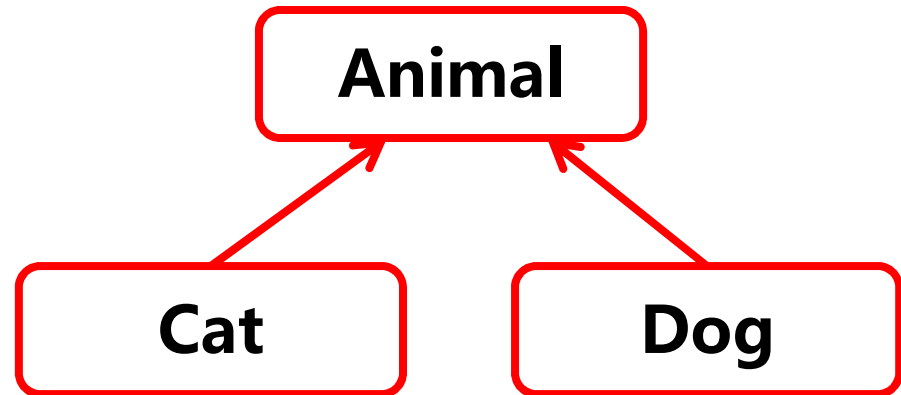
```
Animal a = new Cat();  
Animal b = new Dog();  
Cat c = (Cat)a;
```



多态Polymorphism

□ 动态绑定

```
class Animal {  
    String talk() {  
        return "Error: I am undefined. I don't know how to talk";  
    }  
};  
class Cat extends Animal {  
    String talk() {  
        return "Meow!";  
    }  
};  
class Dog extends Animal {  
    String talk() {  
        return "Woof!";  
    }  
};  
Animal a = new Cat();  
Animal b = new Dog();  
a.talk();  
b.talk();
```



里氏替换法则

Let $q(x)$ be a property provable about objects x of type T . Then $q(y)$ should be true for objects y of type S where S is a subtype of T .

□ Liskov substitution principle

- Barbara Liskov 1987
- 子类对象能够替换其父类对象被使用
- 子类的引用可以直接赋值给父类的引用
- **Animal a = new Cat();**



面向对象

■ 封装

- 设计思想
- 面向数据的封装
- $i = i + j \rightarrow i.add(j.getValue())$

高内聚-低耦合

■ 继承

- 复用

■ 多态

开闭原则

- Liskov替换法则
- 改写-方便复用

■ Simula

- Ole-Johan Dahl and Kristen Nygaard

■ Smalltalk

- Alan Kay

复旦大学计算机科学技术学院



编程方法与技术

7.2. 抽象类、内部类复习

周扬帆

2021-2022第一学期

抽象类

❑ 避免类被错误实例化

■ 强行要求需要继承实现

```
abstract class Animal {  
    public String talk() {  
        return "Error: I am undefined. I don't know how to talk";  
    }  
    ...  
};  
...  
Animal a = new Animal();
```

■ 保证子类一定要实现某个方法：abstract声明

```
abstract class Animal {  
    abstract public String talk()  
    ...  
};
```

内部类

- **定义在其他类定义内部的类**
 - 普通内部类
 - 静态内部类
 - 局部内部类
 - 匿名内部类

普通内部类

- 内部类可使用外部类的变量和方法
- 外部类可以创建内部类实例
- 作用域关键字用法一样
- 需要外部类实例创建内部类对象
- 不能有静态变量
- 可以有常量

外部类没实例化怎么new内部类, 因此

```
public class Log {  
    private String fileName;  
    public class FileLogger() {  
        String a = filename;  
        ...  
    }  
    void WriteLog() {  
        FileLogger fl = new FileLogger();  
        ...  
    }  
}  
  
Log log = new Log();  
Log.FileLogger a = log.new FileLogger();
```



静态内部类

- ❑ 只能访问外部类的static变量和方法
 - 不能直接访问外部类的非static变量和方法
- ❑ 可直接创建，不需要外部类引用
- ❑ 用处
 - 层级非常明显的两个类：提升代码可读性
 - 方便写测试代码

因为外部类没实例化



```
public class Log {  
    private String fileName;  
    public static class Test {  
        ...  
        public static void main(String args[]) {  
            ...  
        }  
    }  
}  
  
Log.Test a = new Log.Test();
```

```

public class TestOrder {
    static {
        System.out.println("1");
    }
    public static class Test {
        static {
            System.out.println("2");
        }
        public static void main(String args[]) {
        }
    }
    public static void main(String argv[]) {
        //Test test;
        System.out.println("3");
    }
}

```

13, java TestOrder

2, java 'TestOrder\$Test'

```

public class TestOrder {
    static {
        System.out.println("1");
    }
    public static class Test {
        static {
            System.out.println("2");
        }
        public static void main(String args[]) {
        }
    }
    public static void main(String argv[]) {
        Test.main(null);
        System.out.println("3");
    }
}

```

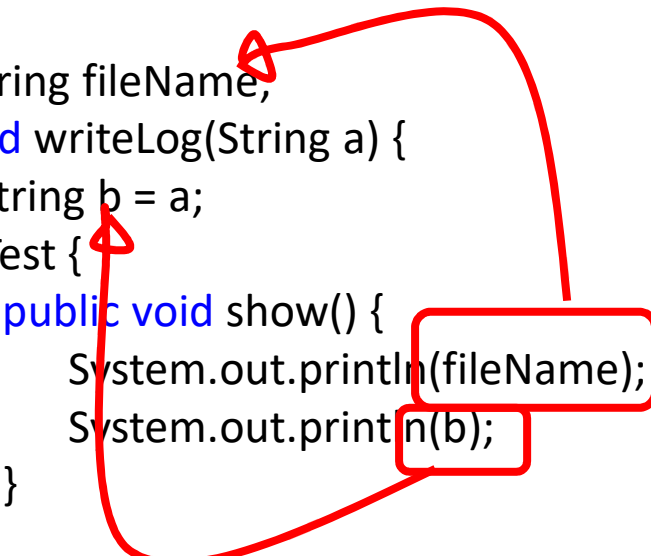
java TestOrder 123

局部内部类

- ❑ 定义在程序块中，只在块内有效
 - 块外不能用到：创建，引用
- ❑ 不加任何访问修饰符，不能加static
 - 但是可以用abstract和final修饰
- ❑ 可访问外部类成员
 - static函数只能访问static外部类成员
- ❑ 可访问块中的final局部变量

1.8, 可以不写，但是必须是effectively final

```
public class Log {  
    private String fileName;  
    public void writeLog(String a) {  
        final String b = a;  
        class Test {  
            public void show() {  
                System.out.println(fileName);  
                System.out.println(b);  
            }  
        }  
        Test a = new Test();  
        a.show();  
    }  
}
```



不是static的方法
的局部内部类

```

public class TestOrder {
    static {
        System.out.println("1");
    }
    static int i = 2;
    public static void main(String argv[]) {
        new TestOrder().test();
    }
    public void test() {
        System.out.println(i);
        class Test extends TestOrder{
            public void test() {
                System.out.println(i);
                i++;
            }
        }
        new Test().test();
        System.out.println(i);
    }
}

```

1
2
2
3

匿名内部类

□ 没有引用名的对象

```
class Test {  
    public void show() {  
        System.out.println("Hello");  
    }  
}
```

父类或接口名字

`new Test().show();`



等价于: `(new Test()).show()`
`new Test()`创建了一个对象
`.show()`调用了这个对象的
`show`方法

□ 匿名类

- 继承父类Test
- 或实现Test接口

`new Test() {`

```
    public void show() {  
        super.show();  
        System.out.println("Hello");  
    }  
}
```

`}.show();`

覆盖了父类的`show()`方法

复旦大学计算机科学技术学院



编程方法与技术

7.3. 接口

周扬帆

2021-2022第一学期

抽象类

□ 抽象类必须被继承实现

■ 可以用来定义某些“共性”

```
abstract class Lecturer {  
    ...  
}  
class Professor extends Lecturer {  
    ...  
}
```

□ 一个类也许有多重不同的“共性”

■ Professor是Lecturer

■ Professor是Researcher

□ Java不支持多重继承

类的共性

- 共性可能不适合抽象为父类
 - 数可以**比较**
 - 字串可以**比较**
 - 某个类的对象可以根据某种规则**比较**
 - 如根据某个成员**比较**

```
abstract class Comparable {  
    abstract boolean compare(Comparable b);  
}  
class Integer extends Comparable {  
    ...  
}
```

接口的定义

```
interface Comparable {  
    String name = "Hello, World";  
    boolean compare(Comparable b);  
}  
  
class MyInteger implements Comparable {  
    public boolean compare(MyInteger b) {  
        ...  
    }  
}
```

- 把需实现的方法和共有常量定义在接口里
 - 接口的变量默认为 **public final static** : 常量
 - 接口的方法默认为 **public abstract**
- 实现接口的类，必须定义接口的方法
 - 接口类似只有 **abstract** 方法和常量的类
 - 不同 (?)

接口的定义

- 同样有public和default两种接口
 - 和class一样
 - public的接口，必须定义在同名文件里
- 同样可以有内部接口
 - 类似内部类，但没有局部接口
- 同样可以用匿名类实现接口

```
interface Printable {  
    void print();  
}
```

```
new Printable() {  
    public void print() { ...  
    }  
}.print();
```

接口的组合

□ 接口可以extends其他接口, 组成新接口

■ 可以extends多个其他接口

```
interface Printable {  
    void print();  
}  
  
interface Printable2 {  
    void print2();  
}  
  
interface Comparable extends Printable, Printable2 {  
    boolean compare(Comparable a);  
}  
  
class Data implement Comparable {  
    public void print() { ...  
    }  
    boolean compare(Comparable a) { ...  
    }  
    public void print2() { ...  
    }  
}
```

实现多接口

□ 一个类可以实现多个接口

```
interface Comparable {  
    String name = "Hello, World";  
    boolean compare(Comparable b);  
}  
interface Printable {  
    void print();  
}  
class MyInteger implements Comparable, Printable {  
    public boolean compare(MyInteger b) {  
        ...  
    }  
    public void print() { ...  
    }  
}
```

逗号分隔

■ 类似多重继承

→ 有没有钻石问题?

接口的默认方法

- ❑ 接口不能实现具体方法非常不方便
- ❑ Java 1.8让接口可以写具体实现

```
interface OldInterface {  
    void a();  
    default void b() {  
        System.out.println("Hello!");  
    }  
}  
  
public class Test implements OldInterface {  
    public static void main(String args[]) {  
        new Test().b();  
    }  
    public void a() {  
        // ...  
    }  
}
```

接口的默认方法

- Java 1.8让接口可以写具体实现
- 二义问题怎么解决?

```
interface OldInterface {  
    void a();  
    default void b() {  
        System.out.println("Hello!");  
    }  
}  
  
interface NewInterface {  
    void a();  
    default void b() {  
        System.out.println("Hello Again!");  
    }  
}  
  
public class Test implements OldInterface, NewInterface {  
    //...  
}
```

需要定义b()的实现!

接口的默认方法

- Java 1.8让接口可以写具体实现
- 二义问题怎么解决?

```
interface OldInterface {  
    void a();  
    default void b() {  
        System.out.println("Hello!");  
    }  
}  
  
interface NewInterface {  
    void a();  
    default void b() {  
        System.out.println("Hello Again!");  
    }  
}  
  
public class Test implements OldInterface, NewInterface {  
    //...  
}
```

需要定义b()的实现! → 如何调用默认定义?

显式调用

OldInterface.super.b();
NewInterface.super.b();

接口的默认方法


- **Java 1.8让接口可以写具体实现**
- **二义问题怎么解决?**
 - **继承的父类有一个同名方法**
 - **实现的接口有一个同名默认方法**
 - **父类优先!**

接口的默认方法

□ 接口升级

□ 如果升级了接口，需要多加方法

```
interface MyInterface {  
    void a();  
}  
  
interface MyInterface {  
    void a();  
    default void b() {  
    }  
}
```



- 好处：老的实现接口的类不用修改，不会报错
- 无法保证新的实现接口的类不会忘记实现b

接口的静态方法

□ 方便复用

```
interface MyInterface {  
    void static a() {  
        // ...  
    }  
}
```

□ 只能通过接口名.方法名调用

MyInterface.a()

- 继承该接口的接口，不能继承静态方法
- 实现该接口的类，也不能继承静态方法

接口的私有方法

□ 方便接口内复用

```
interface MyInterface {  
    private static void a() {  
        // ...  
    }  
    static void b() {  
        a();  
    }  
}
```

```
interface MyInterface {  
    private default void a() {  
        // ...  
    }  
    default void b() {  
        a();  
    }  
}
```

思考

- ❑ 理解接口的目的，在自己日常编程使用接口
- ❑ 比较接口与抽象类，思考什么场景适用接口，什么场景适用抽象类
- ❑ 接口是否可继继承接口？抽象类是否可实现接口？抽象类是否可继继承具体类？
- ❑ Java接口与C++多重继承有什么异同点？
- ❑ 思考你熟悉的其他语言有无（类似）接口的设计

复旦大学计算机科学技术学院



编程方法与技术

7.4. 泛型

周扬帆

2021-2022第一学期

例子: LinkedList

```
import java.util.LinkedList;
public class Test {
    private static void print(LinkedList list) {
        for(int i = 0; i < list.size(); i++) {
            System.out.println(list.get(i));
        }
        System.out.println("-----");
    }
    public static void main(String args[]) {
        LinkedList <String> list = new LinkedList <String>();
        list.add("abc");
        list.add("bcd");
        list.add("cde");
        print(list);
        list.remove(1);
        print(list);
    }
}
```

```
abc
bcd
cde
-----
abc
cde
-----
```


例子: LinkedList

```
import java.util.LinkedList;
public class Test {
    private static void print(LinkedList list) { ...
    }
    public static void main(String args[]) {
        LinkedList <String> list = new LinkedList <String> ();
        list.add("abc");
        list.add("bcd");
        print(list);
        LinkedList<Integer> list2 = new LinkedList<Integer>();
        list2.add(123);
        list2.add(234);
        list2.add(456);
        print(list2);
        list2.remove(1);
        print(list2);
    }
}
```

abc
bcd

123
234
345

123
345

Java泛型类

□ 类的泛型（模板类）

- 类的实现中，把某些用到的数据类型抽象为泛型（模板）
- 在类创建的时候才指定类型
- 此模板可以接受合适类型的对象
- 目的？

```
public class Print <T> {  
    public void print(T [] a) {  
        for(T i: a) {  
            System.out.print(i + " ");  
        }  
        System.out.println();  
    }  
    public static void main(String args[]) {  
        Character [] ca = {'1', '2'};  
        new Print<Character>().print(ca);  
    }  
}
```

Java泛型类

```
class Print <T> {  
    public void print(T [] a) {  
        for(T i: a) {  
            System.out.print(i + " ");  
        }  
        System.out.println();  
    }  
}  
  
public class Test {  
    public static void main(String args[]) {  
        Character [] ca = {'1', '2'};  
        Print<Character> printer = new Print <Character>();  
        printer.print(ca);  
    }  
}
```

定义这个类里有一种类型叫做T,
具体是什么, new的时候才指定

把T作为一种类型,

new的时候指定T是什么

Java泛型类

```
class Print <T> {  
    public void print(T [] a) {  
        for(T i: a) {  
            System.out.print(i + " ");  
        }  
        System.out.println();  
    }  
}  
  
public class Test {  
    public static void main(String args[]) {  
        Character [] ca = {'1', '2'};  
        Print<Character> printer = new Print<Character>();  
        printer.print(ca);  
        Integer[] ca2 = {34, 56};  
        Print<Integer> printer2 = new Print<Integer>();  
        printer2.print(ca2);  
    }  
}
```

1 2
34 56

如果没有泛型

```
class Print {
```

```
    public void print(Integer [] a) {  
        for(Integer i: a) {  
            System.out.print(i + " ");  
        }  
        System.out.println();  
    }
```

```
    public void print(Character [] a) {  
        for(Character i: a) {  
            System.out.print(i + " ");  
        }  
        System.out.println();  
    }
```

代码很雷同

```
}
```

```
public class Test {
```

```
    public static void main(String args[]) {  
        Character [] ca = {'1', '2'};  
        new Print().print(ca);  
        Integer [] ca2 = {34, 56};  
        new Print().print(ca2);  
    }
```

1 2
34 56

```
}
```

不，我有父类

```
class Print {  
    public void print(Object [] a) {  
        for(Object i: a) {  
            System.out.print(i + " ");  
        }  
        System.out.println();  
    }  
}  
  
public class Test {  
    public static void main(String args[]) {  
        Character [] ca = {'1', '2'};  
        new Print().print(ca);  
        Integer [] ca2 = {34, 56};  
        new Print().print(ca2);  
    }  
}
```

Java: Object是所有类的根父类

接受所有类型的数组

1 2
34 56

不，我有父类

```
class Datum {  
    private Object var;  
    public Object getVar() {  
        return var;  
    }  
    public void setVar(Object var2) {  
        var = var2;  
    }  
}  
  
public class Test {  
    public static void main(String args[]) {  
        Datum datum = new Datum();  
        datum.setVar(30);  
        System.out.println(2 * datum.getVar();  
        (Integer) datum.getVar());  
    }  
}
```

setVar把数据存在var
getVar返回数据

60

容易出错

```
class Datum {  
    private Object var;  
    public Object getVar() {  
        return var;  
    }  
    public void setVar(Object var2) {  
        var = var2;  
    }  
}  
public class Test {  
    public static void main(String args[]) {  
        Datum datum = new Datum();  
        datum.setVar("Wrong");  
        System.out.println(2 * (Integer) datum.getVar());  
    }  
}
```

语法符合, 因为String是Object的子类

运行出错, 因为String不能被转成Integer

java.lang.ClassCastException: java.lang.String cannot be cast to java.lang.Integer

Java泛型便于开发时侦错

```
class Datum <T> {  
    private T var;  
    public T getVar() {  
        return var;  
    }  
    public void setVar(T var2) {  
        var = var2;  
    }  
}  
  
public class Test {  
    public static void main(String args[]) {  
        Datum <Integer> datum = new Datum <Integer>();  
        datum.setVar("Wrong");  
        System.out.println(2 * (Integer) datum.getVar());  
    }  
}
```

为什么语法错好过运行错？

语法检查不通过，因为类型不符合

The method setVar(Integer) in the type Datum<Integer> is not applicable for the arguments (String)

Java泛型类

□ 类的泛型（模板类）

- 类的实现中，把某些用到的数据类型抽象为模板
- 在类创建的时候才指定类型
- 此模板可以接受合适类型的对象
- 目的？
 - 相对于使用父类类型，更加安全
 - 编译的时候可以进行类型检查，避免代码写错

Java泛型

```
class Datum <T> {  
    private T var;   
    public T getVar() {  
        return var;  
    }  
    public void setVar(T var2) {  
        var = var2;  
    }  
}
```

← 可做变量定义

← 可做返回值定义

← 可做参数定义

Java泛型

```
class Data <T, S> {  
    private T var1;  
    private S var2;  
    ...  
}
```

← 可定义多个泛型

```
Data <String, String> data1 = new Data <String, String> ();  
Data <String, Integer> data2 = new Data <String, Integer > ();
```

} 不同类型实例

Java泛型

```
class Data <T, S> {  
    private T var1;  
    private S var2;  
    ...  
}
```

□ 常用命名规则

- 采用一个大写字母

- 常用T

- 有多个的话

- 用T附近的, 如S, R等

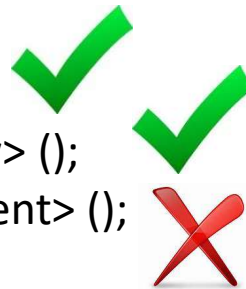
- E – Element, K – Key, V – Value

Java泛型类

```
class Element {  
    ...  
}  
class ElementNew extends Element {  
    ...  
}  
class OtherElement {  
    ...  
}  
class Data <T extends Element> {  
    private T var1;  
    ...  
}
```

← 可限定类型必须是某个指定类型或其子类，或是实现了某个接口，都用extends


```
Data <Element> ele1 = new Data <Element> ();  
Data <ElementNew> ele2 = new Data <ElementNew> ();  
Data <OtherElement> ele3 = new Data <OtherElement> ();
```



什么目的?

Java泛型类

```
class Datum <T> {  
    private T var;  
    public T getVar() {  
        return var;  
    }  
    public void setVar(T var2) {  
        var = var2;  
    }  
    public static void print (T var) {  
        System.out.println(var);  
    }  
}
```



静态方法不允许用类的泛型!
为什么?

和泛型类相似

Java泛型接口

```
interface DatumInterface <S> {  
    public S getVar();  
    public void setVar(S var);  
}
```

```
class Datum <T> implements DatumInterface <T> {  
    private T var;  
    public T getVar() {  
        return var;  
    }  
    public void setVar(T var2) {  
        var = var2;  
    }  
}
```

```
public interface Comparable<T> {  
    public int compareTo(T o)  
}
```

支持泛型实现

Java泛型方法

```
public class Test {  
    public <T> void print(T a) {  
        System.out.println(a);  
    }  
    public static void main(String args[]) {  
        Test test= new Test();  
        test.print("Hello");  
        test.print(12);  
    }  
}
```

定义这个方法有一种类型叫做T，具体是什么，调用的时候指定

在调用时，指定方法里的T为String

在调用时，指定方法里的T为Integer

Java泛型方法

```
public class Test {  
    public static <T> void print(T a) {  
        System.out.println(a);  
    }  
    public static void main(String args[]) {  
        print("Hello");  
        print(12);  
    }  
}
```

支持静态方法



静态方法不允许用类的泛型 → 为什么静态方法可以实现为泛型方法？

Java泛型方法：更多例子

```
public class Test {  
    static <T extends Comparable<T>> int compare(T a, T b) {  
        if (a.compareTo(b) > 0) {  
            return 1;  
        }  
        else if (a.compareTo(b) < 0) {  
            return -1;  
        }  
        return 0;  
    }  
    public static void main(String args[]) {  
        System.out.println("ret = " + compare(12, 13));  
        System.out.println("ret = " + compare("ABC", "abc"));  
    }  
}
```

指定T必须实现
Comparable泛型接口
也就是T要有compareTo
方法

```
public interface Comparable<T> {  
    public int compareTo(T o)  
}
```

Java泛型类

```
class Datum <T> {  
    private T var;  
    public T getVar() {  
        return var;  
    }  
    public void setVar(T var2) {  
        var = var2;  
    }  
}
```

```
Datum <String> datum1 = new Datum <String> ();  
Datum <Object> datum2 = new Datum < Object > ();
```

Fact: Object是所有类的根父类

Question: Datum <Object>是Datum <String>的父类吗?

```
datum2.setVar("Hello");  
datum2.setVar(20);
```



```
datum1.setVar("Hello");  
datum1.setVar(20);
```



■ Liskov替换法则

不能创建 (new) 泛型变量和数组

```
class Datum <T> {  
   private T a1 [] = new T [10];  
  private T a2 = new T();  
}
```

为了代码安全而禁止

编译器不知道如何初始化
万一T需要带参数初始化?

a1[0] = "String"; (类型擦除 运行时无法检查a1是什么类型数组)
如果T不是String, 就容易出错

→ 禁止泛型数组, 避免上述错误

```
Integer a [] = new Integer [10];  
Object b [] = a;  
允许, 因为Object 是 Integer 的父类  
b[0] = "String";
```

没有泛型对象数组

```
class Datum <T> {
```

```
    ...
```

```
}
```

```
Datum <String> a [] = new Datum <String> [10];
```



- Fact: Java泛型只在编译时处理，运行时是擦除的

- 为了兼容性

- 为了方便

- 因此

```
class Datum <T> {
```

```
    ...
```

```
}
```

```
Datum <String> a [] = new Datum <String> [10];
```

```
a[0] = new Datum <Integer>();
```

- 容易出各种错

常用Java泛型类

□ ArrayList

视为变长数组

```
import java.util.ArrayList;

...
ArrayList<Integer> list = new ArrayList<Integer> ();
int i = 0;
for(i = 0; i < 10; i++ ) {
    //给数组增加10个Int元素
    list.add(i);
}
list.remove (5); //将第6个元素移除
for(i = 0; i < 3; i++ ) {
    //再增加3个元素
    list.add(i+20);
}
i = list.get(3); //得到第4个元素
Integer t[] = new Integer[list.size()]; //list.size()得到当前大小
list.toArray(t); //转成数组
```

常用Java泛型类

□ LinkedList

视为链表

```
import java.util.LinkedList;

...
LinkedList<Integer> list = new LinkedList<Integer>();
int i = 0;
for(i = 0; i < 10; i++ ) {
    //给数组增加10个Int元素
    list.add(i);
}
list.remove(5); //将第6个元素移除
for(i = 0; i < 3; i++ ) {
    //再增加3个元素
    list.add(i+20);
}
i = list.get(3); //得到第4个元素
Integer t[] = new Integer[list.size()]; //list.size()得到当前大小
list.toArray(t); //转成数组
```

和前面一样

思考

- ❑ 思考Java泛型和C++模板的区别
- ❑ 理解泛型中的各种继承
- ❑ 理解虚拟机如何处理Java泛型，理解类型擦除
- ❑ 自行理解List <? super T>里super的作用
- ❑ 可以把List<String>传递给一个接受List<Object>参数的方法吗？

复旦大学计算机科学技术学院



编程方法与技术

7.5. 课堂练习

周扬帆

2021-2022第一学期

LinkedList 泛型类

□ 自己实现LinkedList 泛型类

- **public void** addLast(E e)
 - 在linkedlist末尾添加元素
- **public E** removeLast()
 - 返回链表的末尾一个元素，并删除它
- **public boolean** contains(E e)
 - 返回true，如果LinkedList中包含此element
- **public E** get(int index)
 - 返回在index处的element
- **public int** size()
 - 返回大小

Stack泛型类

□ 使用自己实现的LinkedList泛型类实现

- **public E pop()**: 移除栈顶部的对象，并作为此函数的值返回该对象
- **public E push(E e)**: 把element压入栈顶部
- **public E peek()**: 查看栈顶部的对象，但不从堆栈中移除它
- **public boolean empty()**: 判断栈是否为空

□ 实现测试程序

- 使用自己的Stack类，把泛型实例化为Integer、String和某种自己定义的类型，比如Test
- 测试push/pop/peek/empty方法

学有余力:实现图

- **用LinkedList实现一个动态图的数据结构**
 - 每个节点用一个LinkedList表示, 里面存邻居, 用一个泛型表示节点的标示(ID)
 - 图用一个ListedList存节点列表
- **实现一个这样的图的类**
 - 自己设计构造方法和其他方法
 - 目的: 方便初始化一个图
- **实现一个hasPath方法, 判断两个节点是否有路径**
- **实现简单测试程序**