面向对象的软件构造导论

第4讲 接口与继承

课程（回顾）

- 对象与类
- 类的声明与构造
- 类的访问域
- static修饰符
- 数组

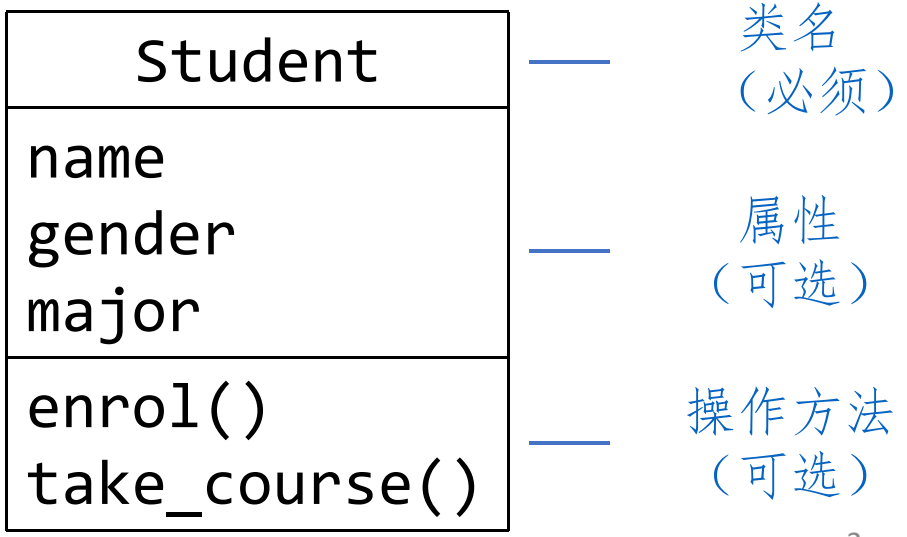
对象与类

对象(Object)：客观存在的具体实体，具有明确定义的状态和行为

类(Class)：对现实生活中一类具有共同属性和共同操作的对象的抽象

举例

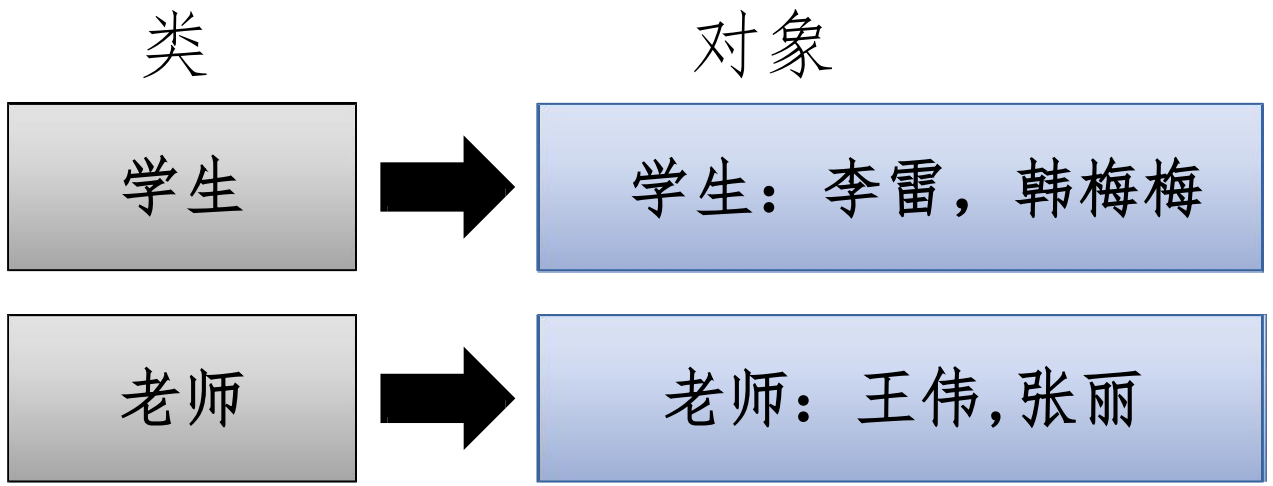
- 类名：学生
  - 有属性：姓名，性别，专业；
  - 有操作：被录取，选课



类与对象的对比

类与对象之间的关系

- 类是对象的抽象，是创建对象的模板（代表了同一批对象的共性与特征）
- 对象是类的具体实例（不同对象之间还存在着差异）
- 同一个类可以定义多个对象（一对多关系）



类的声明

- 格式

```
[类修饰符] class 类名
{
        成员变量的声明; // 描述属性
        成员方法的声明; // 描述功能
 }
```

类修饰符
- public：公共类
- abstract：抽象类(继承)
- final：最终类(非继承)

- 举例

```
public class Person {
      String name;
      int age;

      setName(String name) {...}
      getName() {...}
}
```

类的访问域

- 类的访问权限控制

	private	*default*	protected	public
同一个类中				
同一个包中				
子类中				
全局范围内				

包(package)：我们在java编程中经常把功能相似或者相关的类放在一个包里

类的访问域

类的访问权限控制

- public：该类可以被其他类所访问
- *default*：该类只能被同一个包中的类访问
- private：无法被其他类所访问
- protected：可以被子类访问，以及子类的子类（作用于继承关系）

	private	*default*	protected	public
同一个类中	√	√	√	√
同一个包中		√	√	√
子类中			√	√
全局范围内				√

作用：
实现封装特性

包(package)：我们在java编程中经常把功能相似或者相关的类放在一个包里

类的访问域

- 成员的访问权限控制?

```
package p;
public class Demo{
        private int var1=1;
        int var2 = 2;
        protected int var3 = 3;
        public int var4 = 4;
              …
}
```

	var1	var2	var3	var4
package p1				
package p				
class newDemo1 extends Demo in p				
class newDemo2 extends Demo in p1				
class Demo				

静态成员

含义

- 表明该属性、该方法是属于类的，称为静态属性或静态方法

（无static修饰，则是实例属性或实例方法）

```
static 成员属性;      // 静态属性
static 成员方法;      // 静态方法
```

说明

- 静态成员属于类所有，不属于某一具体对象私有；
- 静态成员随类加载时被静态地分配内存空间、方法的入口地址

静态成员

- 注意
  - 使用static声明的方法，不能访问非static的操作（属性或方法）
  - 非static声明的方法，可以访问static声明的属性或方法
- 原因
  - 如果一个类中的属性和方法都是非static类型的，一定要有实例化对象才可以调用
  - Static声明的属性或方法可以通过类名访问，可以在没有实例化对象的情况下调用

数组

多维数组的使用：初始化

- 动态初始化：int[][] arr = new int[m][n];
  - 二维数组中有m个一维数组，每个一维数组中有n个元素
- 动态初始化：int[][] arr = new int[m][];
  - 二维数组中有m个一维数组，每个一维数组都是默认初始化值null
  - 可以对这个三个一维数组分别进行初始化

  arr[0] = new int[3]; arr[1]= new int[1]; arr[2] = new int[2];

  - int[ ][ ]arr = new int[ ][3] // 非法！

数组

多维数组的使用：初始化

- 静态初始化：

```
int[][] arr = new int[][]{{1,2,3},{2,7},{4,5,6,7}};
```

  - 定义一个名称为arr的二维数组，二维数组中有三个一维数组
  - arr[0] = {1,2,3}; arr[1]= {2,7}; arr[2] = {4,5,6,7};
  - 特殊写法情况: int[] x, y[]; x是一维数组，y是二维数组
  - Java中多维数组不必都是规则矩阵形式

数组

对象数组

- 数组中的元素是对象，数组中的每一个元素都是对一个对象的引用

```
Person[] students;

Person students[];
```

```
class Person {
    private String name;
    private int age;
    // …
    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    // …
}
```

数组

- 对象数组静态初始化
  - 在定义数组的同时对数组元素进行初始化

```
Person[] students = {
    new Person(“张三”, 21),
    new Person(“李四”, 19)
}
```

数组

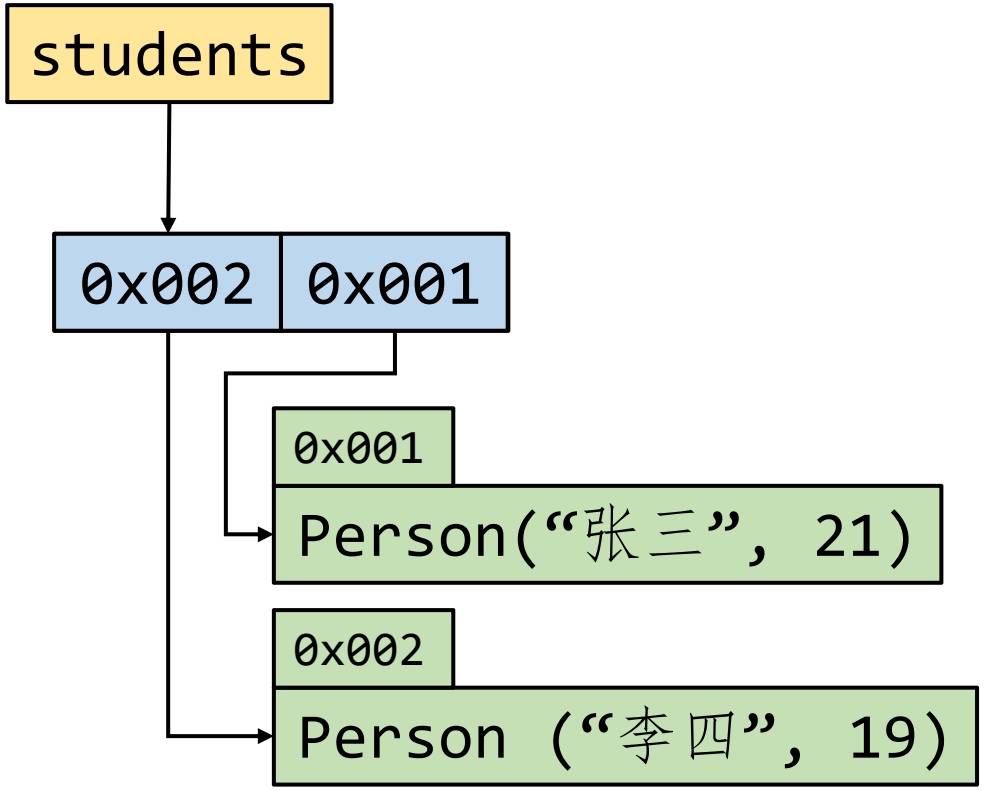
- 对象数组动态初始化
  - 使用运算符new为数组分配空间

```
Person[] students = new Person[2];
Person per1 = new Person(“张三”, 21);
Person per2 = new Person(“李四”, 19);
students[0] = per1;
students[1] = per2;
```

数组

- 对象数组动态初始化

```
Person[] students = new Person[2];
Person per1 = new Person(“张三”, 21);
Person per2 = new Person(“李四”, 19);
students[0] = per2;
students[1] = per1;
```



课程内容

- 继承
- 接口与抽象类
- 多态与重写
- Java多继承问题
- 超类与super关键字
- 异常的继承框架

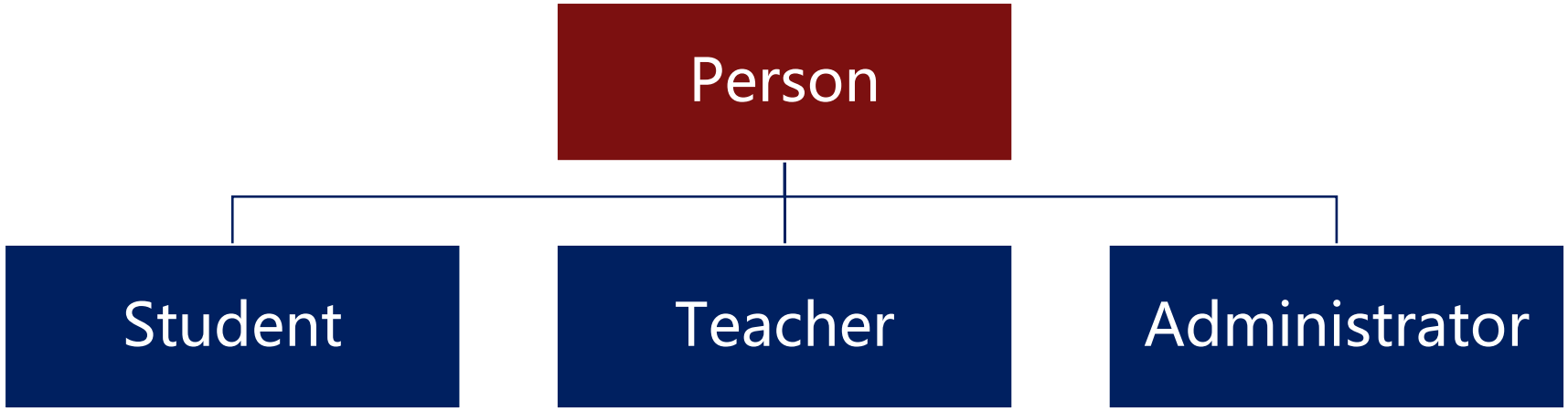
为什么要继承？

```
class Person {
    String name;
    int age;
    public String getName() {...}
    public void setName(String name) {...}
}
```

```
class Student {
    String name;
    int age;
    String school;
    public String getName() {...}
    public void setName(String name) {...}
    public String getSchool() {...}
}
```

大量重复的代码！

继承-父类和子类



- Person类与其他类(如Student类) 存在关联
- 在Student与Person之间存在着明显的is-a-kind-of关系，这是**继承**的一个明显特征

继承-类，超类和子类

- 关键词**extends**表明正在构造的新类派生于一个已存在的类。
  - 已存在的类：超类(superclass)，基类，父类
  - 新类：子类(subclass)，派生类，孩子类
- 继承能**复用**已存在的类的方法，并增加一些新的方法和字段，使得新类能够适应新的情况。
- 父类/超类刻画子类所**共同拥有的属性和方法**。
- 不同的子类有各自不同的属性和方法

继承-定义父类、子类

```
1. public class Person{
2.     private String name;
3.     private int age;
4.     public Person(String name, int age){
5.         this.name = name;
6.         this.age = age;
7.     }
8.     public String getName(){
9.         return name;
10.    }
11.    private String setName (String name){
12.        this.name = name;
13.    }
14.}
```

```
1. public class Student extends Person{
2.     String school;
3.     public Student(String name, int age,
   String school) {
4.      super(name, age); // 调用父类的构造方法
5.      this.school= school;
6.     }
7.
8.
9.     public string getSchool(){
10.        return this.school;
11.    }
12.}
```

- 利用extends关键字
- 子类不能直接继承父类的构造方法，需利用super关键字
- 子类能**复用**父类的属性和方法，子类能**增加**一些新的属性与方法

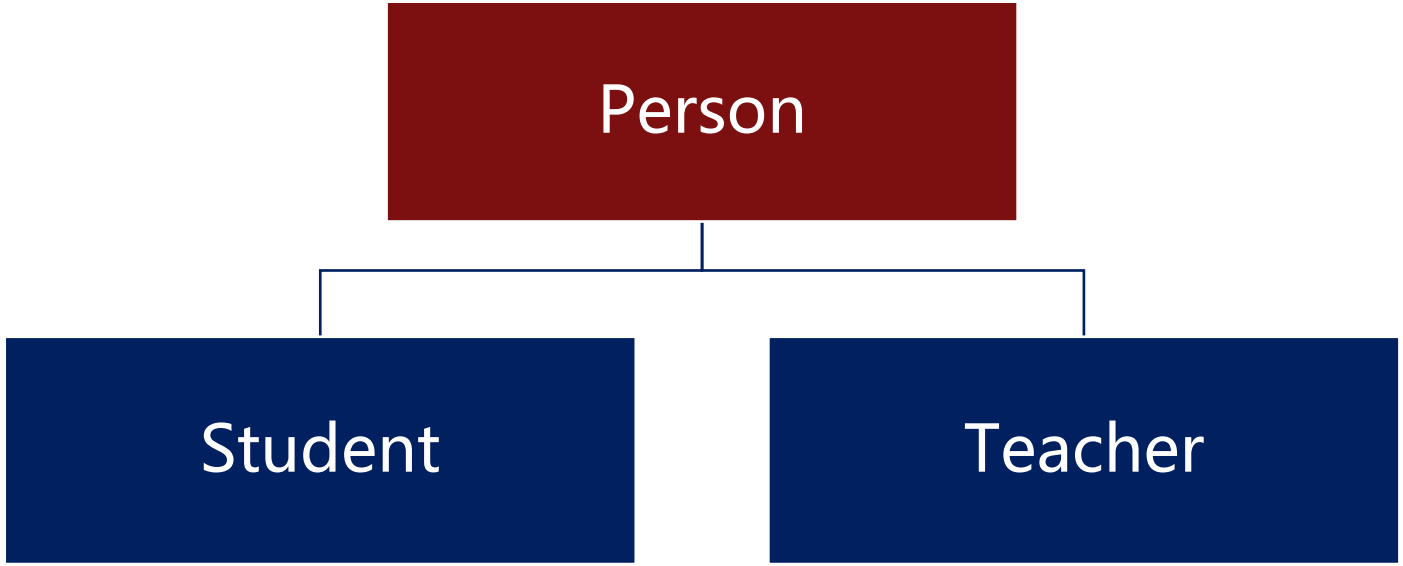
继承

- 继承是面向对象编程的三大重要特性之一
  - 继承与发展
- 优点
  - 提高代码的可复用性
  - 提高程序的扩展性
  - 使类与类之间产生了关系，构成了**多态**的基础
- 缺点
  - 让类的耦合性增强（一个类的改变会影响到其他相关类）

课程内容

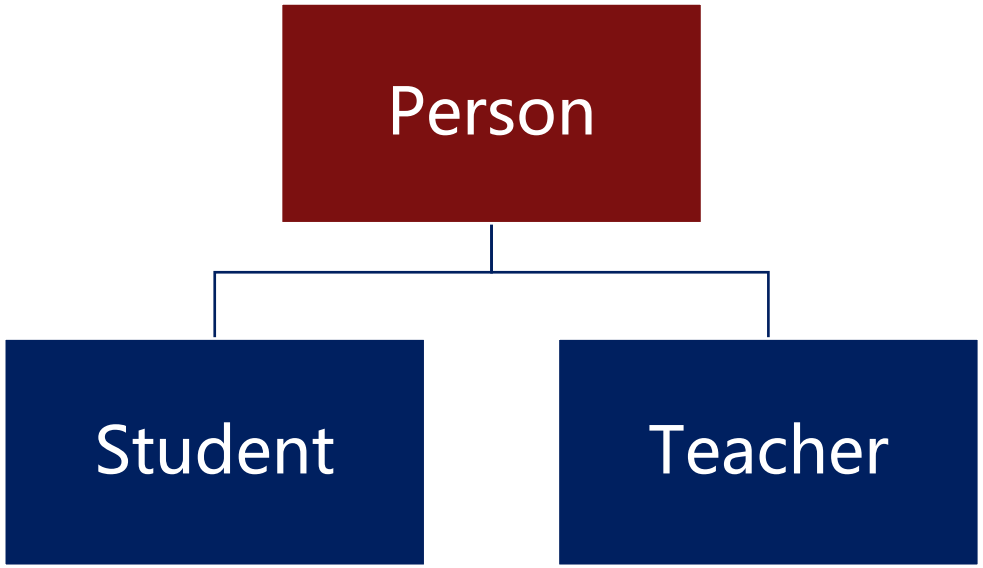
- 继承
- 接口与抽象类
- 多态与重写
- Java多继承问题
- 超类与super关键字
- 异常的继承框架

抽象类-定义抽象类



- 父类是将子类所共同拥有的属性和方法进行抽取。
- 这些属性和方法中，如果有的还无法确定，那么我们就可以将其定义成**抽象(abstract)方法**，在后面子类进行具体化实现。
- 简单说，拥有抽象方法的类就是**抽象类**。

抽象类-定义抽象类




```
1. //定义一个抽象类
2. public abstract class Person{
3.     //普通方法
4.     public String getName()
5.     {
6.         return name;
7.     }
8.     //抽象方法
9.     //没有方法体，用abstract做修饰
10.     public abstract void getMission();
11.}
12.
13.Person P = new Person();
14.// 编译错误，抽象类不能实例化
```

抽象类-抽象类的子类

- 抽象方法充当着占位方法的角色，在子类中具体实现

```
public class Student extends Person{
    public void getMission(){
        System.out.println("为中华之崛起而读书");
    }
}
```

```
public class Teacher extends Person{
    public void getMission(){
        System.out.println("立德树人");
    }
}
```

抽象类-抽象类的作用

- 抽象类是不能实例化的
- 作用：
  - 抽象方法实际上相当于定义了“规范”
  - **只能被继承**，保证子类实现其定义的抽象方法
  - 可用于实现**多态**

接口

- 抽象类中，抽象方法本质上是定义接口规范，保证子类都有相同的接口实现
- **接口(interface)：**
  - 比抽象类还要抽象：没有字段，所有方法都是抽象方法

```
1. //定义一个抽象类
2. public abstract class Person{
3.     //抽象方法
4.     public abstract void getDuty();
5.     public abstract void getMission();
6. }
```

```
1. //定义一个接口
2. interface Person{
3.     //抽象方法
4.     void getDuty();
5.     void getMission();
6. }
```

接口

- 当一个具体的class去实现一个interface时，使用implements关键字

```
public class Student implements Person{
    private String name;
    private int age;
    @Override
    public void getDuty(){
        System.out.println("好好学习");
    }
    @Override
    public void getMission(){
        System.out.println("为中华之崛起而读书");
    }
}
```

接口 VS. 抽象类

- 接口与抽象类比较

	Abstract class	Interface
抽象方法	可以定义抽象方法	可以定义抽象方法
字段	可以定义字段	无字段
继承	只能extends一个class	可以implements多个interface

- 思考：有了抽象类，为什么还需要接口？
  - 抽象类解决不了多继承的问题
  - 要实现的方法不是当前类的必要方法
    - 例如“会唱歌” 不是Person类的必要方法，如果设置成抽象方法会浪费资源
  - 为不同类型的多个类实现同样的方法
    - 例如：Person、鸟类、收音机、手机都能“唱歌”

课程内容

- 继承
- 接口与抽象类
- 多态与重写
- Java多继承问题
- 超类与super关键字
- 异常的继承框架

多态

- 多态（Polymorphism）是面向对象编程的三大重要特性之一
- 多态是同一个行为具有多种表现形态的能力
  - 不同情况下的不同处理方式
  - 程序中定义的变量和方法在编程时并不确定，而是在程序运行期间才确定。

多态

- 多态的好处
  - 减少耦合
  - 增强可替换性
  - 增强可扩展性
  - 提高灵活性

- 使用多态的三个必要条件：
  - 继承
  - 重写
  - 父类引用指向子类

- 多态的三种实现方式：
  - 重写
  - 抽象类和抽象方法
  - 接口

多态的实现1：重写

- 父类中的某些方法对子类并不一定适用。需要**重写/覆盖**(override)父类中的这个方法。
- 重写：如果在子类中定义一个方法，其**名称、参数、返回类型**正好与父类中某个方法相同，那么可以说，子类的方法**重写**了父类的方法。

多态的实现1：重写

```
1. public class Person{
2.     private String name;
3.     private int age;

4.     public void getTarget(){
5.         System.out.println( “美好生活” );
6.     }
7. }
```

```
1. public class Student extends Person{
2.     @Override
       public void getTarget(){
3.         System.out.println(“功夫到家");
4.     }
5. }
```

```
1. public class Teacher extends Person{
2.     @Override
       public void getTarget(){
3.         System.out.println(“传道授业解惑");
4.     }
5. }
```

- 在Student和Teacher类中，我们重写了getTarget方法，覆盖之前的getTarget.
- 这些方法有相同的名称、返回类型及参数。
- 编程技巧：加上@Override可以让编译器帮助检查是否进行了正确的覆写。

多态的实现1：重写

```
public static void show_target(Person P_x) {
     System.out.print(" 目标: ");
     P_x.getTarget();
 }
Person P1= new Student();
Person P2= new Teacher();
show_target(P1);
show_target(P2);
P1.getTarget();
P2.getTarget();
```

- 通过父类的引用调用子类重写的方法
- 在show_target()函数中我们只需要传入父类的引用
  - 父类类名 引用名称 = new 子类类名();

System.out.println(P1.getClass()); // 输出:?

多态的实现1：重写

```
public static void show_target(Person P_x) {
     System.out.print(" 目标: ");
     P_x.getTarget();
 }
Person P1= new Student();
Person P2= new Teacher();
show_target(P1);
show_target(P2);
P1.getTarget();
P2.getTarget();
```

- 通过父类的引用调用子类重写的方法
- 在show_target()函数中我们只需要传入父类的引用
  - 父类类名 引用名称 = new 子类类名();

System.out.println(P1.getClass()); // 输出: class Student

多态的实现2：抽象类

```
1. //定义一个抽象类
2. public abstract class Person{
3.     ……
4.     //抽象方法
5.     //没有方法体，用abstract做修饰
6.     public abstract void getTarget();
7.     ……
8. }
```

多态的实现2：抽象类

```
public class Engineer extends Person{
    @Override
    public void getTarget(){
        System.out.println(“技术改变世界");
    }
}
```

```
public class Scientist extends Person{
    @Override
    public void getTarget(){
        System.out.println(“勇攀科学高峰");
    }
}
```

```
public static void main(String[] args) {
    Person Wang = new Engineer();
    Wang.getTarget();
    Person Li = new Scientist();
    Li.getTarget();
}
```

- 多态的定义格式：父类类名 引用名称 = new 子类类名();

```
Person Wang = new Engineer();
```

多态的实现3：接口

```
//接口：目标
public interface GetTarget(){
    void getTarget();
}
```

接口(interface)是比一般抽象类还要抽象的纯抽象类，没有属性，只有方法。

多态的实现3：接口

```
//Talent类实现了目标接口
public class Talent implements GetTarget(){
    @Override
    public void getTarget(){
        System.out.println(“打造国之重器");
    }
}
```

```
// Entrepreneur类实现了目标接口
public class  Entrepreneur implements GetTarget(){
    @Override
    public void getTarget(){
        System.out.println(“促进共同富裕");
    }
}
```

多态的实现3：接口

```
public static void main(String[] args) {
    GetTarget  Liu = new Talent();
    Liu.getTarget();
    GetTarget  Ma = new Entrepreneur();
    Ma.getTarget();
}
```

这里的对象Liu 和 Ma 都可以看做是接口GetTarget的子类

单选题 1分

设置

下面哪个描述是错误的?

A 继承是多态的基础

B 继承有利于降低类的耦合性，一个类的改变不会影响其他类

C 多态可以提高程序的灵活性

D 接口中一般无属性字段，可定义抽象方法，支持多继承

提交

单选题 1分

设置

下面哪个描述是错误的?

A 继承是多态的基础

B 继承有利于降低类的耦合性，一个类的改变不会影响其他类

C 多态可以提高程序的灵活性

D 接口中一般无属性字段，可定义抽象方法，支持多继承

提交

课程内容

- 继承
- 接口与抽象类
- 多态与重写
- Java多继承问题
- 超类与super关键字
- 异常的继承框架

Java允许多继承吗

- 多继承指的是一个类可以同时从多个父类那里继承行为和特征，然而Java为了保证数据安全，**只允许单继承**。
- 多继承存在的问题：
  - 若子类继承的父类中**拥有相同的成员变量**，子类在引用该变量时将无法判别使用哪个父类的成员变量。
  - 若一个子类继承的**多个父类拥有相同方法**，同时子类并未覆盖该方法（若覆盖，则直接使用子类中该方法），那么调用该方法时将无法确定调用哪个父类的方法。

Java如何实现多继承效果

- 有时候开发人员确实需要实现**多重继承**，而且现实生活中真正地存在这样的情况。例如遗传，我们可以既继承父亲的行为和特征，又继承母亲的行为和特征。
- Java提供的两种方法让我们实现多继承：
  - 内部类
  - 接口

多继承1：内部类

- 内部类：在类内部不仅可以**定义成员变量和方法**，还可以**定义另一个类**。如果在类A的内部再定义一个类B，则**B类就称为内部类**，而**A类则称为外部类**。
  - 内部类可以实现很好的隐藏，并且拥有外部类的所有元素的**访问权限**。

多继承1：内部类实现

- 我们以生活中常见的遗传例子进行介绍，子女是如何利用内部类来实现同时继承父亲和母亲的优良基因的。
- 我们创建Father类，在该类中添加strong()方法；同样的，创建Mother类，在该类中添加smart()方法，代码如下：

```
public class Father{
    public int strong(){
        // 强壮指数
        return 9;
    }
}
```

```
public class Mother{
    public int smart(){
        // 聪慧指数
        return 8;
    }
}
```

多继承1：内部类实现

- 然后，创建Son类，在该类中通过内部类实现多继承效果。代码如右侧所示：
  - Son类定义两个内部类，这两个内部类分别继承Father类和Mother类，且都可以获取各自父类的行为。

```
1. public class Son {
2.     // 内部类继承Father类
3.     class Father_Inner extends Father {
4.         public int strong() {
5.             return super.strong() + 1;
6.         }
7.     }
8.     // 内部类继承Mother类
9.     class Mother_Inner extends Mother {
10.        public int smart() {
11.            return super.smart() + 2;
12.        }
13.    }
14.    public int getStrong() {
15.        return new Father_Inner().strong();
16.    }
17.    public int getSmart() {
18.        return new Mother_Inner().smart();
19.    }
20. }
```

多继承2：用接口实现

- 首先，定义接口Father和接口Mother，然后使用接口Daughter继承了接口Father、Mother。
  - Java 中类只能继承一个类，但是接口可以继承多个接口。

```
1. public interface Father{
2.     public void strong();
3. }
4.
5. public interface Mother{
6.     public void smart();
7. }
8.
9. public interface Daughter extends Father, Mother{
10.     public void kind();
11.}
```

多继承2：用接口实现

然后，类Girl实现了接口Daughter，并在类中**重写**了Father的strong方法、Mother的smart方法以及Daughter的kind方法。

```
1. public class Girl implements Daughter{
2.     public static void main(String[] args){
3.     }
4.     @Override
5.     public void strong(){
6.         System.out.println(“She’s not strong.");
7.     }
8.     @Override
9.     public void smart(){
10.         System.out.println(“She’s very smart.");
11.     }
12.     @Override
13.     public void kind(){
14.         System.out.println(“She’s very kind.")
15.     }
16.}
```

课程内容

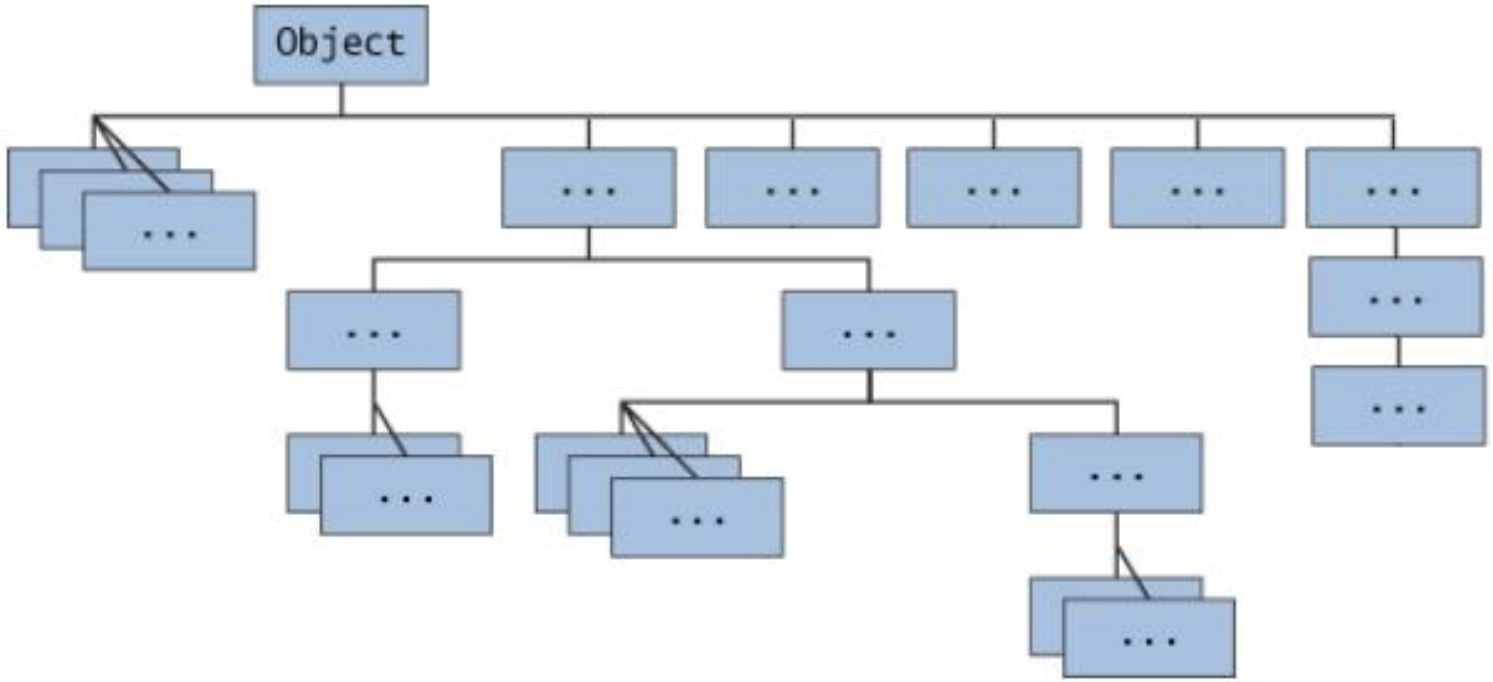
- 继承
- 接口与抽象类
- 多态与重写
- Java多继承问题
- 超类与super关键字
- 异常的继承框架

Object超类

- Object类是Java中所有类的始祖，在Java中每个类都扩展了Object.

  子类可以使用Object的所有方法。

- Object 类位于 java.lang 包中，编译时会自动导入。



Object超类

- Object类可以**显式**继承，也可以**隐式**继承，两种方式均可：
  - 显示继承

```
1. public class Student extends Object{
2.
3. }
```

  - 隐式继承

```
1. public class Student{
2.
3. }
```

Object类型的变量

- 可以用Object类型的变量引用任何的对象：

```
Object obj = new Student(“Zhangsan", 19);
```

- 当然，Object类型的变量只能用于作为各种值的一个泛型容器。要想对其中的内容进行具体的操作，还需要清楚对象的原始类型，并进行相应的强制类型转换：

```
Student Zhangsan = (Student) obj;
```

Object类型的变量

```
Object obj = new Student("Zhangsan", 19);
```

Object 类本身提供了一些基本方法

```
obj.toString()    // 返回对象的字符串表示
obj.equals()      // 比较对象是否相等
obj.hashCode()    // 返回对象的哈希码
obj.getClass()    // 获取对象的实际类型
```

不能直接调用的方法

```
// 这些会编译错误（除非强制转换后）：
obj.getName()  // Student 类的特定方法
obj.getAge()   // Student 类的特定方法
// 任何子类特有的方法
```

强制转换成 *Student* 类后才能使用*Student*类的方法

```
Student Zhangsan = (Student) obj;
```

equals方法

- Object类中的equals()方法用于比较两个对象是否相等。
- equals()方法比较两个对象，是判断两个对象引用是否指向的是同一个对象。

```
1. class IfStudentEqual {
2.     public static void main(String[] args) {
3.         // 创建两个对象
4.         Object Student_1 = new Student("Zhangsan",19);
5.         Object Student_2 = new Student("Lisi",18);
6.
           // 不同对象，内存地址不同，不相等，返回 false
7.         System.out.println(Student_1.equals(Student_2));
8.
           // 对象引用，内存地址相同，相等，返回 true
9.         Object Student_3 = Student_1;
10.        System.out.println(Student_1.equals(Student_3));
11.    }
12.}
```

equals方法

```
Student castedStudent = (Student) Student_3;
System.out.println(Student_1.equals(castedStudent));
```

// 输出:?

equals方法

```
Student castedStudent = (Student) Student_3;
System.out.println(Student_1.equals(castedStudent));
```

// 输出: true

- 强制转换不会改变对象本身：强制转换只是改变了引用的类型，不会改变实际指向的对象。
- equals() 方法比较的是对象本身

super关键字

- super 关键字的功能：
  - 在子类的构造方法中显式地**调用父类构造方法**。
  - 访问父类的**成员方法和变量**。

super调用父类构造方法

- super关键字可以在子类的构造方法中显式地**调用父类的构造方法**，

  基本格式如：super(parameter-list);

  - 其中，parameter-list 指定了父类构造方法中的所有参数。

```
1. public class Person {
2.     public Person(String name, int age) {
3.     }
4. }
5. public class Student extends Person {
6.     public Student(String name, int age, String school) {
7.         super(name, age); // 调用父类中含有2个参数的构造方法
8.     }
9. }
```

super访问父类成员

- 当子类的成员变量或方法与父类**同名**时，例如子类重写了父类的某一个方法，我们可以通过 super关键字来调用父类里面的这个方法。
- 使用 super 访问父类中的成员**与 this 关键字**的使用相似，只不过它引用的是子类的父类，语法格式如：`super.X`
  - 其中，X是父类中的属性或方法。

Super访问父类成员变量

- 在右图的例子中，父类和子类都有一个成员变量age。我们可以使用 super 关键字访问Person类中的age 变量。

```
1.  class Person {
2.      int age = 39;
3.  }
4.  class Student extends Person {
5.      int age = 18;
6.      void display() {
7.          System.out.println("学生年龄: " + super.age);
8.      }
9.  }
10. class Test {
11.     public static void main(String[] args) {
12.         Student stu = new Student();
13.         stu.display();
14.     }
15. }
```

输出是？ ⇨ 学生年龄： 39

super调用父类成员方法

- 当父类和子类都具有相同的方法名时，可以使用 super 关键字访问父类的方法。

```
1.  class Person {
2.      void getTarget() {
3.          System.out.println(“美好生活");
4.      }
5.  }
6.  class Student extends Person {
7.      void getTarget() {
8.          System.out.println("功夫到家");
9.      }
10.     void display() {
11.         getTarget()
12.         super.getTarget();
13.     }
14. }
```

调用当前Student类的getTarget() 方法

调用父类Person类的getTarget() 方法

课程内容

- 继承
- 接口与抽象类
- 多态与重写
- Java多继承问题
- 超类与super关键字
- 异常的继承框架

什么是异常？

- 程序运行时，发生的**不被期望的事件**，它阻止了程序按照程序员的预期正常执行，这就是异常。
  - 情形1：希望用户输入一个int类型的年龄，但是用户输入的却是abc；

```
1.String s = "abc";   // 假设用户输入了abc
2.int n = Integer.parseInt(s); // NumberFormatException!
```

  - 情形2：程序想要读写某个文件的内容，但是用户已经把它删除了；

```
1.// 用户删除了该文件：
2.String t = readFile("C:\\abc.txt"); // FileNotFoundException!
```

Java异常处理机制

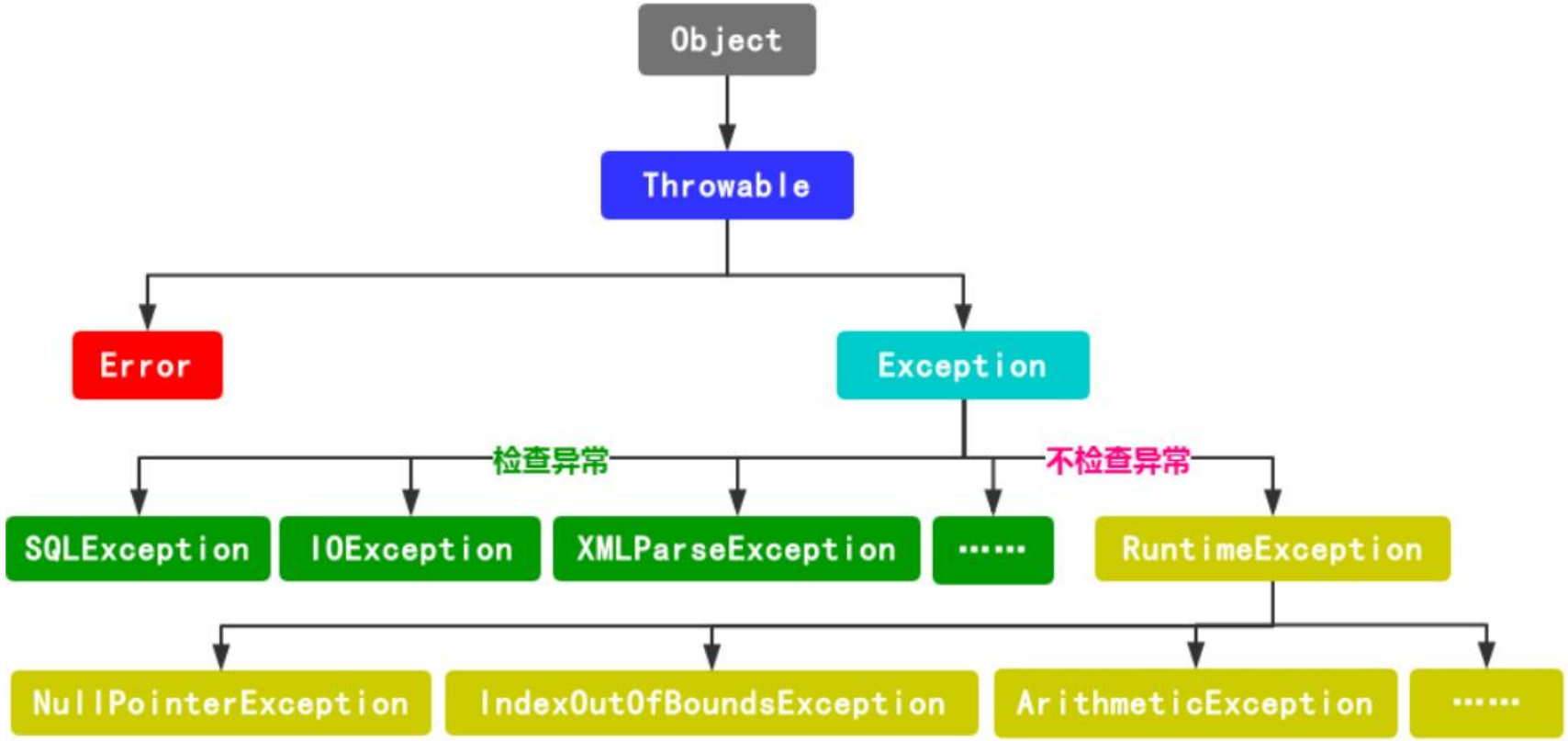
- Java内置了一套**异常处理机制**，总是使用异常来表示错误。
- 异常是一种class，因此它本身带有类型信息。
- 三种类型的异常：
  - **检查性异常**：最具代表的检查性异常是用户错误或问题引起的异常，这是程序员无法预见的。例如要打开一个不存在文件时，一个异常就发生了，这些异常在编译时不能被简单地忽略。
  - **运行时异常**：运行时异常是可能被程序员避免的异常。与检查性异常相反，运行时异常可以在编译时被忽略。
  - **错误**：错误不是异常，而是脱离程序员控制的问题。错误在代码中通常被忽略。例如，当栈溢出时，一个错误就发生了，它们在编译也检查不到的。

使用异常的好处

- 告诉我们程序在哪里出错了。
- 使本来已经中断的程序以适当的方式继续运行，或者退出。
- 保存用户的当前操作，把占用的资源释放掉。
- 降低代码复杂度，把错误和业务代码分离。

异常的继承框架

- 从继承关系可知：Throwable是异常体系的根，它继承自Object。
- 所有的异常类是从java.lang.Exception类继承的子类。



检查性异常和运行时异常的核心区别就在于编译时是否会受到编译器的强制检查

异常的继承框架

- Throwable有两个体系：Error 和 Exception，**Error**表示严重的错误，程序对此一般无能为力。
- 而**Exception**则是运行时的错误，它可以被捕获并处理。Exception类又分为两大类：
  - RuntimeException以及它的子类；
  - 非RuntimeException(包括SQLException、IOException、XMLParseException等)

设计自定义异常

- Java中可以自定义异常。编写自定义异常类时需注意：
  - 所有异常类都必须是Throwable的子类。
  - 如果希望自定义一个检查性异常类，则需要继承Exception类。
  - 如果希望自定义一个运行时异常类，那么需要继承RuntimeException类。
- 可以像下面这样**自定义异常类**：

```
// 自定义异常类
class MyException extends Exception{

}
```

自定义异常实例

- 编写一个程序，对计入系统的成绩进行检查，是否在[0,100]区间。
- 这里首先创建了一个异常类 MyException，并提供两个构造方法。
  - 无参的默认构造方法。
  - 以字符串的形式接收一个定制的异常消息，并将该消息传递给超类的构造方法

```
1. public class MyException extends Exception {
2.     public MyException() {
3.         super();
4.     }
5.     public MyException(String str) {
6.         super(str);
7.     }
8. }
```

自定义异常实例

- 接着创建测试类，调用自定义异常类。

```
1. import java.util.InputMismatchException;
2. import java.util.Scanner;

3. public class exception_eg {
4.     public static void main(String[] args) {
5.         int score;
6.         Scanner input = new Scanner(System.in);
7.         System.out.println("请输入你的考试成绩：");
8. ……
```

自定义异常实例

```
9.          try {
10.             score = input.nextInt();     // 获取成绩
11.             if(score < 0) {
12.                 throw new MyException("成绩为负！");
13.             } else if(score > 100) {
14.                 throw new MyException("你的优秀已经溢出了！");
15.             } else {
16.                 System.out.println("你的成绩为："+score);
17.             }
18.         } catch(InputMismatchException e1) {
19.             System.out.println("输入的成绩不是数字！");
20.         } catch(MyException e2) {
21.             System.out.println(e2.getMessage());
22.         }
23.……
```

自定义异常实例

- 运行该程序，当用户输入的分数为负数时，则抛出MyException自定义异常，执行第二个catch语句块中的代码，打印出异常信息。程序的运行结果如下所示。

```
请输入你的的成绩：
101
你的优秀已经溢出了！
```

课程内容

- 继承
- 接口与抽象类
- 多态与重写
- Java多继承问题
- 超类与super关键字
- 异常的继承框架

参考

- Java核心技术卷I 第5章 第7章
- 廖雪峰Java教程
  https://www.liaoxuefeng.com