day03【多态、代码块、包、权限修饰符、内部类，Object类】

今日内容

- 多态
- 代码块
- 包
- 权限修饰符
- 内部类
- Object类

教学目标

- [ ] 能够说出使用多态的前提条件
- [ ] 理解多态的向上转型
- [ ] 理解多态的向下转型
- [ ] 能够写出静态代码块的格式
- [ ] 描述final修饰的类的特点
- [ ] 描述final修饰的方法的特点
- [ ] 能够说出权限修饰符作用范
- [ ] 说出内部类的概念
- [ ] 能够说出Object类的特点
- [ ] 能够重写Object类的toString方法
- [ ] 能够重写Object类的equals方法

第一章 多态

1.1 多态的形式

多态是继封装、继承之后，面向对象的第三大特性。

多态是出现在继承或者实现关系中的。

多态体现的格式：

```
父类类型 变量名 = new 子类/实现类构造器;
变量名.方法名();
```

多态的前提：有继承关系，子类对象是可以赋值给父类类型的变量。例如Animal是一个动物类型，而Cat是一个猫类型。Cat继承了Animal，Cat对象也是Animal类型，自然可以赋值给父类类型的变量。

1.2 多态的案例演示

当使用多态方式调用方法时，首先检查父类中是否有该方法，如果没有，则编译错误；如果有，执行的是子类重写后方法。如果子类没有重写该方法，就会调用父类的该方法。

总结起来就是：**编译看左边，运行看右边。**

代码如下：

定义父类：

```
public class Animal {
    public void eat() {
        System.out.println("动物吃东西！")
    }
}
```

定义子类：

```
class Cat extends Animal {
    public void eat() {
        System.out.println("吃鱼");
    }
}

class Dog extends Animal {
    public void eat() {
        System.out.println("吃骨头");
    }
}
```

定义测试类：

```
public class Test {
    public static void main(String[] args) {
        // 多态形式，创建对象
        Animal a1 = new Cat();
        // 调用的是 Cat 的 eat
        a1.eat();

        // 多态形式，创建对象
        Animal a2 = new Dog();
        // 调用的是 Dog 的 eat
        a2.eat();
    }
}
```

1.3 多态的定义和前提

多态：是指同一行为，具有多个不同表现形式。

从上面案例可以看出，Cat和Dog都是动物，都是吃这一行为，但是出现的效果（表现形式）是不一样的。

前提【重点】

1. 继承或者实现【二选一】
2. 方法的重写【意义体现：不重写，无意义】
3. 父类引用指向子类对象【格式体现】

> 父类类型：指子类对象继承的父类类型，或者实现的父接口类型。

1.4 多态的好处

实际开发的过程中，父类类型作为方法形式参数，传递子类对象给方法，进行方法的调用，更能体现出多态的扩展性与便利。代码如下：

定义父类：

```
public abstract class Animal {
    public abstract void eat();
}
```

定义子类：

```
class Cat extends Animal {
    public void eat() {
        System.out.println("吃鱼");
    }
}

class Dog extends Animal {
    public void eat() {
        System.out.println("吃骨头");
    }
}
```

定义测试类：

```
public class Test {
    public static void main(String[] args) {
        // 多态形式，创建对象
        Cat c = new Cat();
        Dog d = new Dog();

        // 调用showCatEat
        showCatEat(c);
        // 调用showDogEat
        showDogEat(d);

        /*
        以上两个方法，均可以被showAnimalEat(Animal a)方法所替代
```

```
        而执行效果一致
        */
        showAnimalEat(c);
        showAnimalEat(d);
    }

    public static void showCatEat (Cat c){
        c.eat();
    }

    public static void showDogEat (Dog d){
        d.eat();
    }

    public static void showAnimalEat (Animal a){
        a.eat();
    }
}
```

由于多态特性的支持，showAnimalEat方法的Animal类型，是Cat和Dog的父类类型，父类类型接收子类对象，当然可以把Cat对象和Dog对象，传递给方法。

当eat方法执行时，多态规定，执行的是子类重写的方法，那么效果自然与showCatEat、showDogEat方法一致，所以showAnimalEat完全可以替代以上两方法。

不仅仅是替代，在扩展性方面，无论之后再多的子类出现，我们都不需要编写showXxxEat方法了，直接使用showAnimalEat都可以完成。**从而实现了实现类的自动切换。**

所以，多态的好处，体现在，可以使程序编写的更简单，并有良好的扩展。

1.5 多态的弊端

我们已经知道多态编译阶段是看左边父类类型的，如果子类有些独有的功能，此时**多态的写法就无法访问子类独有功能了。**

```
class Animal{
    public  void eat() {
        System.out.println("动物吃东西！")
    }
}
class Cat extends Animal {
    public void eat() {
        System.out.println("吃鱼");
    }

    public void catchMouse() {
        System.out.println("抓老鼠");
    }
}

class Dog extends Animal {
    public void eat() {
        System.out.println("吃骨头");
```

```
    }
}

class Test{
    public static void main(String[] args){
        Animal a = new Cat();
        a.eat();
        a.catchMouse();//编译报错，编译看左边，Animal没有这个方法
    }
}
```

1.6 引用类型转换

1.6.1 为什么要转型

多态的写法就无法访问子类独有功能了。

当使用多态方式调用方法时，首先检查父类中是否有该方法，如果没有，则编译错误。也就是说，**不能调用**子类拥有，而父类没有的方法。编译都错误，更别说运行了。这也是多态给我们带来的一点"小麻烦"。所以，想要调用子类特有的方法，必须做向下转型。

回顾基本数据类型转换

- 自动转换: 范围小的赋值给范围大的.自动完成:double d = 5;
- 强制转换: 范围大的赋值给范围小的,强制转换:int i = (int)3.14

多态的转型分为向上转型（自动转换）与向下转型（强制转换）两种。

1.6.2 向上转型（自动转换）

- **向上转型**：多态本身是子类类型向父类类型向上转换（自动转换）的过程，这个过程是默认的。 当父类引用指向一个子类对象时，便是向上转型。 使用格式：

```
父类类型   变量名 = new 子类类型();
如：Animal a = new Cat();
```

原因是：父类类型相对与子类来说是大范围的类型，Animal是动物类，是父类类型。Cat是猫类，是子类类型。Animal类型的范围当然很大，包含一切动物。所以子类范围小可以直接自动转型给父类类型的变量。

1.6.3 向下转型（强制转换）

- **向下转型**：父类类型向子类类型向下转换的过程，这个过程是强制的。 一个已经向上转型的子类对象，将父类引用转为子类引用，可以使用强制类型转换的格式，便是向下转型。

使用格式：

```
子类类型 变量名 = （子类类型） 父类变量名;
如:Aniaml a = new Cat();
   Cat c =(Cat) a;
```

1.6.4 案例演示

当使用多态方式调用方法时，首先检查父类中是否有该方法，如果没有，则编译错误。也就是说，**不能调用**子类拥有，而父类没有的方法。编译都错误，更别说运行了。这也是多态给我们带来的一点"小麻烦"。所以，想要调用子类特有的方法，必须做向下转型。

转型演示，代码如下：

定义类：

```
abstract class Animal {
    abstract void eat();
}

class Cat extends Animal {
    public void eat() {
        System.out.println("吃鱼");
    }
    public void catchMouse() {
        System.out.println("抓老鼠");
    }
}

class Dog extends Animal {
    public void eat() {
        System.out.println("吃骨头");
    }
    public void watchHouse() {
        System.out.println("看家");
    }
}
```

定义测试类：

```
public class Test {
    public static void main(String[] args) {
        // 向上转型
        Animal a = new Cat();
        a.eat();               // 调用的是 Cat 的 eat

        // 向下转型
        Cat c = (Cat)a;
        c.catchMouse();        // 调用的是 Cat 的 catchMouse
    }
}
```

1.6.5 转型的异常

转型的过程中，一不小心就会遇到这样的问题，请看如下代码：

```
public class Test {
    public static void main(String[] args) {
        // 向上转型
        Animal a = new Cat();
        a.eat();               // 调用的是 Cat 的 eat

        // 向下转型
        Dog d = (Dog)a;
        d.watchHouse();        // 调用的是 Dog 的 watchHouse 【运行报错】
    }
}
```

这段代码可以通过编译，但是运行时，却报出了 `ClassCastException` ，类型转换异常！这是因为，明明创建了Cat类型对象，运行时，当然不能转换成Dog对象的。

1.6.6 instanceof关键字

为了避免ClassCastException的发生，Java提供了 `instanceof` 关键字，给引用变量做类型的校验，格式如下：

```
变量名 instanceof 数据类型
如果变量属于该数据类型或者其子类类型，返回true。
如果变量不属于该数据类型或者其子类类型，返回false。
```

所以，转换前，我们最好先做一个判断，代码如下：

```
public class Test {
    public static void main(String[] args) {
        // 向上转型
        Animal a = new Cat();
        a.eat();               // 调用的是 Cat 的 eat

        // 向下转型
        if (a instanceof Cat){
            Cat c = (Cat)a;
            c.catchMouse();        // 调用的是 Cat 的 catchMouse
        } else if (a instanceof Dog){
            Dog d = (Dog)a;
            d.watchHouse();       // 调用的是 Dog 的 watchHouse
        }
    }
}
```

第二章 内部类

2.1 概述

2.1.1 什么是内部类

将一个类A定义在另一个类B里面，里面的那个类A就称为**内部类**，B则称为**外部类**。可以把内部类理解成寄生，外部类理解成宿主。

内部类是Java类的五大成份之一，也是我们最后一个需要学习的成份。

2.1.2 什么时候使用内部类

一个事物内部还有一个独立的事物，内部的事物脱离外部的事物无法独立使用

1. 人里面有一颗心脏。
2. 汽车内部有一个发动机。
3. 为了实现更好的封装性。

2.2 内部类的分类

按定义的位置来分

1. **静态内部类**，类定义在了成员位置 (类中方法外称为成员位置，有static修饰的内部类)
2. **实例内部内**，类定义在了成员位置 (类中方法外称为成员位置，无static修饰的内部类)
3. **局部内部类**，类定义在方法内
4. **匿名内部类**。一般定义在方法中，或者可执行代码中

2.3 静态内部类

静态内部类特点:

- 有static修饰的内部类，属于外部类本身的。
- 总结：静态内部类与其他类的用法完全一样。只是访问的时候需要加上外部类.内部类。
- **拓展**:静态内部类可以直接访问外部类的静态成员。

内部类的使用格式:

```
外部类.内部类。
```

静态内部类对象的创建格式:

```
外部类.内部类  变量 = new  外部类.内部类构造器;
```

案例演示:

```
// 外部类: Outer01
class Outer01{

    private static  String sc_name = "黑马程序";

    // 内部类: Inner01
    public static class Inner01{
        // 这里面的东西与类是完全一样的。
        private String name;

        public Inner01(String name) {
```

```
            this.name = name;
        }

        public void showName(){
            System.out.println(this.name);
            // 拓展:静态内部类可以直接访问外部类的静态成员。
            System.out.println(sc_name);
        }
    }
}

public class InnerClassDemo01 {
    public static void main(String[] args) {
        // 创建静态内部类对象。
        // 外部类.内部类  变量 = new  外部类.内部类构造器;
        Outer01.Inner01 in  = new Outer01.Inner01("张三");
        in.showName();
    }
}
```

2.4 实例内部类

实例内部类特点:

- 无static修饰的内部类，属于外部类对象的。
- 宿主：外部类对象。

内部类的使用格式:

```
外部类.内部类。 // 访问内部类的类型都是用 外部类.内部类
```

实例内部类创建对象格式:

```
外部类.内部类 变量 = new 外部类构造器.new 内部类构造器;
```

- 拓展1：实例内部类不能定义静态成员。
- 拓展2：实例内部类可以直接访问外部类的私有和静态成员。

  案例演示

```
public class InnerClassDemo02 {
    public static void main(String[] args) {
        //  宿主：外部类对象。
       // Outer02 out = new Outer02();
        // 创建内部类对象。
        Outer02.Inner02 in = new Outer02().new Inner02("张三");
        in.showName();
    }
}
```

```
class Outer02{

    // 实例内部类，属于外部类对象的。
    // 拓展：实例内部类不能定义静态成员。
    public class Inner02{
        // 这里面的东西与类是完全一样的。
        private String name;

        public Inner02(String name) {
            this.name = name;
        }

        public void showName(){
            System.out.println(this.name);
        }
    }
}
```

2.5 实例内部类面试题

请在?地方向上相应代码,以达到输出的内容

注意：内部类访问外部类对象的格式是：**外部类名.this**

```
public class Demo05 {
    public static void main(String[] args) {
        Body.Heart heart = new Body().new Heart();
        heart.jump();
    }
}

class Body {    // 身体
    private int weight = 30;

    // 在成员位置定义一个类
    class Heart {
        private int weight = 20;

        public void jump() {
            int weight = 10;
            System.out.println("心脏在跳动 " + ?);   // 10
            System.out.println("心脏在跳动 " + ?);   // 20
            System.out.println("心脏在跳动 " + ?);   // 30
        }
    }
}
```

2.6 局部内部类

- **局部内部类**： 定义在**方法中**的类。

定义格式:

```
class 外部类名 {
    数据类型 变量名;

    修饰符 返回值类型 方法名(参数列表) {
        // …
        class 内部类 {
            // 成员变量
            // 成员方法
        }
    }
}
```

> 局部内部类编译后仍然是一个独立的类，编译后有$还有一个数字。Chinese$1Chopsticks.class

2.7 匿名内部类【重点】

2.7.1 概述

匿名内部类：是内部类的简化写法。它的本质是一个`带具体实现的` `父类或者父接口的` `匿名的` **子类对象。** 开发中，最常用到的内部类就是匿名内部类了。

2.7.2 引入

实际上，如果我们希望定义一个只要使用一次的类，就可考虑使用匿名内部类。匿名内部类的本质作用是为了简化代码。

之前我們使用接口时，似乎得做如下几步操作：

1. 定义子类
2. 重写接口中的方法
3. 创建子类对象
4. 调用重写后的方法

```
interface Swim {
    public abstract void swimming();
}

// 1. 定义接口的实现类
class Student implements Swim {
    // 2. 重写抽象方法
    @Override
    public void swimming() {
        System.out.println("狗刨式...");
    }
}

public class Demo07 {
    public static void main(String[] args) {
        // 3. 创建实现类对象
        Student s = new Student();
        // 4. 调用方法
```

```
        s.swimming();
    }
}
```

我们的目的，最终只是为了调用方法，那么能不能简化一下，把以上四步合成一步呢？匿名内部类就是做这样的快捷方式。

2.7.3 匿名内部类前提和格式

匿名内部类必须**继承一个父类**或者**实现一个父接口。**

匿名内部类格式

```
new 父类名或者接口名(){
    // 方法重写
    @Override
    public void method() {
        // 执行语句
    }
};
```

2.7.4 使用方式

以接口为例，匿名内部类的使用，代码如下：

创建匿名内部类，并调用：GUI做界面

```
interface Swim {
    public abstract void swimming();
}

public class Demo07 {
    public static void main(String[] args) {
        // 使用匿名内部类
        new Swim() {
            @Override
            public void swimming() {
                System.out.println("自由泳...");
            }
        }.swimming();

        // 接口 变量 = new 实现类(); // 多态,走子类的重写方法
        Swim s2 = new Swim() {
            @Override
            public void swimming() {
                System.out.println("蛙泳...");
            }
        };

        s2.swimming();
        s2.swimming();
    }
```

```
}
```

2.7.5 匿名内部类的特点

1. 定义一个没有名字的内部类
2. 这个类实现了父类，或者父类接口
3. 匿名内部类会创建这个没有名字的类的对象

2.7.6 匿名内部类的使用场景

通常在方法的形式参数是接口或者抽象类时，也可以将匿名内部类作为参数传递。代码如下：

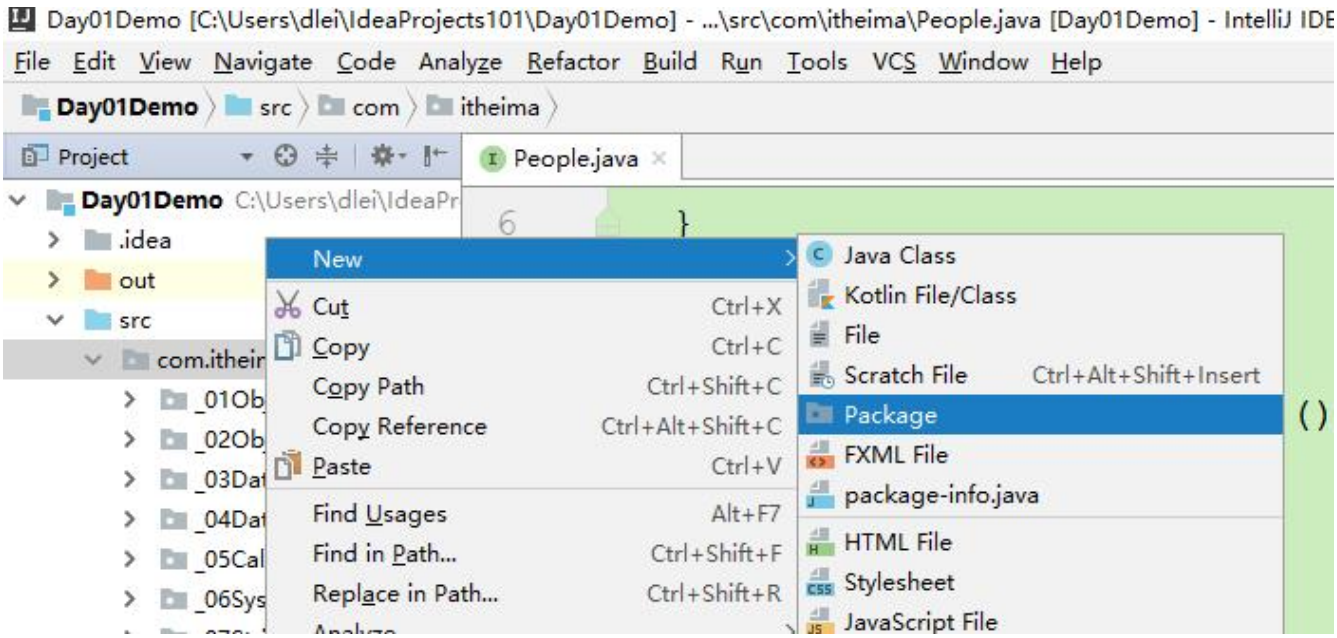
```
interface Swim {
    public abstract void swimming();
}

public class Demo07 {
    public static void main(String[] args) {
        // 普通方式传入对象
        // 创建实现类对象
        Student s = new Student();

        goSwimming(s);
        // 匿名内部类使用场景:作为方法参数传递
        Swim s3 = new Swim() {
            @Override
            public void swimming() {
                System.out.println("蝶泳...");
            }
        };
        // 传入匿名内部类
        goSwimming(s3);

        // 完美方案：一步到位
        goSwimming(new Swim() {
            public void swimming() {
                System.out.println("大学生，蛙泳...");
            }
        });

        goSwimming(new Swim() {
            public void swimming() {
                System.out.println("小学生，自由泳...");
            }
        });
    }

    // 定义一个方法,模拟请一些人去游泳
    public static void goSwimming(Swim s) {
        s.swimming();
    }
}
```

第三章 包和权限修饰符

3.1 包

包我们每天建的项目就是在一个目录下，我们每次都会建立一个包，这个包在磁盘下其实就是一个目录。**包是用来分门别类的管理技术，不同的技术类放在不同的包下**，方便管理和维护。

在IDEA项目中，建包的操作如下：



包名的命名规范：

```
路径名.路径名.xxx.xxx
// 例如：com.itheima.oa
```

- 包名一般是公司域名的倒写。例如：黑马是www.itheima.com,包名就可以定义成com.itheima.技术名称。
- 包名必须用"."连接。
- 包名的每个路径名必须是一个合法的标识符，而且不能是Java的关键字。

3.2 权限修饰符

在Java中提供了四种访问权限，使用不同的访问权限修饰符修饰时，被修饰的内容会有不同的访问权限，我们之前已经学习过了public 和 private，接下来我们研究一下protected和缺省（default默认）修饰符的作用。

- public：公共的，所有地方都可以访问。
- protected：当前类，当前包，当前类的子类可以访问。
- 缺省（没有修饰符）：当前类，当前包可以访问。
- private：私有的，当前类可以访问。 `public > protected > 缺省 > private`

3.3 不同权限的访问能力

	public	protected	缺省（空的）	private
同一类中	√	√	√	√
同一包中的类	√	√	√	
不同包的子类	√	√		
不同包中的无关类	√			

可见，public具有最大权限。private则是最小权限。

编写代码时，如果没有特殊的考虑，建议这样使用权限：

- 成员变量使用`private`，隐藏细节。
- 构造方法使用`public`，方便创建对象。
- 成员方法使用`public`，方便调用方法。

> 小贴士：不加权限修饰符，就是default权限

第四章 代码块

4.1 引入

类有且仅有五大成分： 1.成员变量 2.构造器 3.成员方法 4.代码块 5.内部类

我们已经学完了成员变量，构造器，成员方法，接下来我们来介绍以下代码快，代码块按照有无static可以分为静态代码块和实例代码块。

4.2 静态代码块

静态代码块 必须有static修饰，必须放在类下。与类一起加载执行。

格式

```
static{
     // 执行代码
}
```

特点：

- 每次执行类，加载类的时候都会先执行静态代码块一次。
- 静态代码块是自动触发执行的，只要程序启动静态代码块就会先执行一次。
- 作用：在启动程序之前可以做资源的初始化，一般用于初始化静态资源。

案例演示

```
public class DaimaKuaiDemo01 {
    public static String sc_name ;
```

```
    // 1.静态代码块
    static {
        // 初始化静态资源
        sc_name = "黑马程序员! ";
        System.out.println("静态代码块执行! ");
    }

    public static void main(String[] args) {
        System.out.println("main方法执行");
        System.out.println(sc_name);
    }
}
```

4.3 实例代码块

实例代码块 没有static修饰，必须放在类下。与对象初始化一起加载。

格式

```
{
     // 执行代码
}
```

特点:

- 无static修饰。属于对象，与对象的创建一起执行的。
- 每次调用构造器初始化对象，实例代码块都要自动触发执行一次。
- 实例代码块实际上是提取到每一个构造器中去执行的。
- 作用：实例代码块用于初始化对象的资源。

案例演示

```
public class DaimaKuaiDemo02 {

    private String name ;

    // 实例代码块。 无static修饰。
    {
        System.out.println("实例代码块执行");
        name = "dl";
    }

    // 构造器
    public DaimaKuaiDemo02(){
        //System.out.println("实例代码块执行");
    }

    // 有参数构造器
    public DaimaKuaiDemo02(String name){
        //System.out.println("实例代码块执行");
    }
```

```
    public static void main(String[] args) {
        // 匿名对象，创建出来没有给变量。
        new DaimaKuaiDemo02();
        new DaimaKuaiDemo02();
        new DaimaKuaiDemo02("xulei");
    }
}
// 输出三次：实例代码块执行
```

第五章 Object类

5.1 概述

`java.lang.Object`类是Java语言中的根类，即所有类的父类。它中描述的所有方法子类都可以使用。在对象实例化的时候，最终找的父类就是Object。

如果一个类没有特别指定父类， 那么默认则继承自Object类。例如：

```
public class MyClass /*extends Object*/ {
    // ...
}
```

根据JDK源代码及Object类的API文档，Object类当中包含的方法有11个。今天我们主要学习其中的2个：

- `public String toString()`：返回该对象的字符串表示。
- `public boolean equals(Object obj)`：指示其他某个对象是否与此对象“相等”。

5.2 toString方法

方法摘要

- `public String toString()`：返回该对象的字符串表示。

toString方法返回该对象的字符串表示，其实该字符串内容就是对象的类型+@+内存地址值。

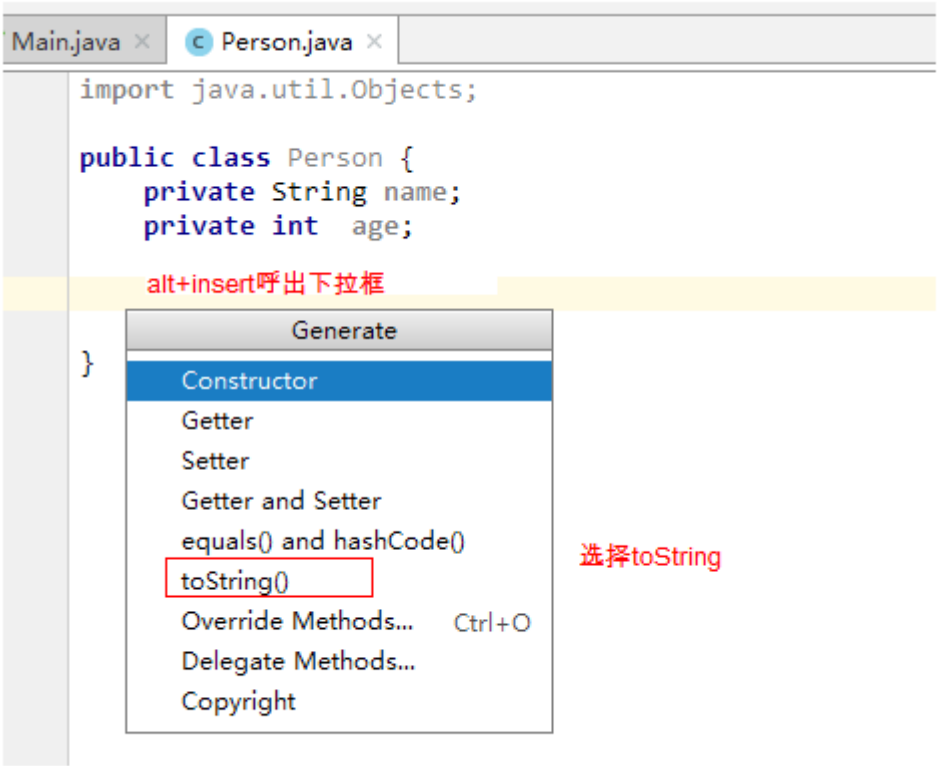
由于toString方法返回的结果是内存地址，而在开发中，经常需要按照对象的属性得到相应的字符串表现形式，因此也需要重写它。

覆盖重写

如果不希望使用toString方法的默认行为，则可以对它进行覆盖重写。例如自定义的Person类：

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

    @Override
    public String toString() {
        return "Person{" + "name='" + name + '\'' + ", age=" + age + '}';
    }

    // 省略构造器与Getter Setter
}
```

在IntelliJ IDEA中，可以点击 Code 菜单中的 Generate... ，也可以使用快捷键 alt+insert ，点击 toString() 选项。选择需要包含的成员变量并确定。如下图所示：



> 小贴士： 在我们直接使用输出语句输出对象名的时候,其实通过该对象调用了其toString()方法。

5.3 equals方法

方法摘要

- `public boolean equals(Object obj)`：指示其他某个对象是否与此对象“相等”。

调用成员方法equals并指定参数为另一个对象，则可以判断这两个对象是否是相同的。这里的“相同”有默认和自定义两种方式。

默认地址比较

如果没有覆盖重写equals方法，那么Object类中默认进行`==`运算符的对象地址比较，只要不是同一个对象，结果必然为false。

对象内容比较

如果希望进行对象的内容比较，即所有或指定的部分成员变量相同就判定两个对象相同，则可以覆盖重写equals方法。例如：

```
import java.util.Objects;

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

    @Override
    public boolean equals(Object o) {
        // 如果对象地址一样，则认为相同
        if (this == o)
            return true;
        // 如果参数为空，或者类型信息不一样，则认为不同
        if (o == null || getClass() != o.getClass())
            return false;
        // 转换为当前类型
        Person person = (Person) o;
        // 要求基本类型相等，并且将引用类型交给java.util.Objects类的equals静态方法取用结果
        return age == person.age && Objects.equals(name, person.name);
    }
}
```

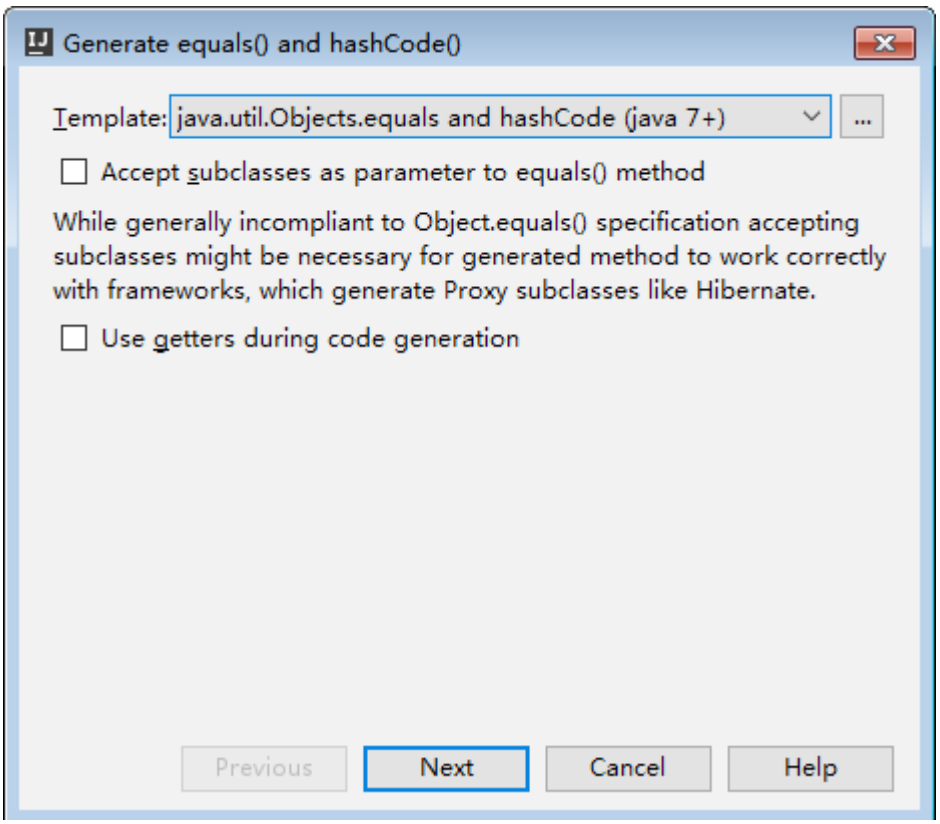
这段代码充分考虑了对象为空、类型一致等问题，但方法内容并不唯一。大多数IDE都可以自动生成equals方法的代码内容。在IntelliJ IDEA中，可以使用`Code`菜单中的`Generate...`选项，也可以使用快捷键`alt+insert`，并选择`equals() and hashCode()`进行自动代码生成。如下图所示：

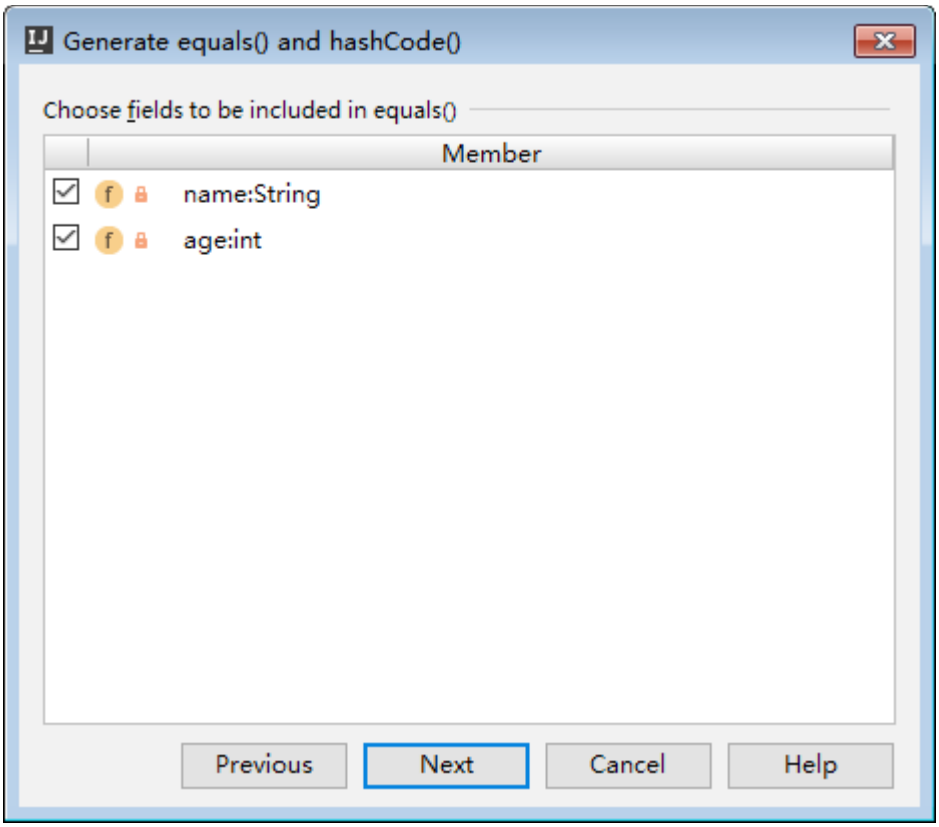
Main.java Person.java

```
1  import java.util.Objects;
2
3  public class Person {
4      private String name;
5      private int  age;
6
7
8
9
10 }
11
```

Generate

- Constructor
- Getter
- Setter
- Getter and Setter
- equals() and hashCode()
- toString()
- Override Methods... Ctrl+O
- Delegate Methods...
- Copyright





> tips：Object类当中的hashCode等其他方法，今后学习。

第六章 Objects类

Objects类是对象工具类，它里面的的方法都是用来操作对象的。

6.1 equals方法

在刚才IDEA自动重写equals代码中，使用到了`java.util.Objects`类，那么这个类是什么呢?

在**JDK7**添加了一个Objects工具类，它提供了一些方法来操作对象，它由一些静态的实用方法组成，这些方法是null-save（空指针安全的）或null-tolerant（容忍空指针的），用于计算对象的hashcode、返回对象的字符串表示形式、比较两个对象。

在比较两个对象的时候，Object的equals方法容易抛出空指针异常，而Objects类中的equals方法就优化了这个问题。方法如下：

- `public static boolean equals(Object a, Object b)`:判断两个对象是否相等。

我们可以查看一下源码，学习一下：

```
public static boolean equals(Object a, Object b) {
    return (a == b) || (a != null && a.equals(b));
}
```

6.2 isNull

static boolean isNull(Object obj) 判断对象是否为null，如果为null返回true。

```
Student s1 = null;
Student s2 = new Student("蔡徐坤", 22);

// static boolean isNull(Object obj) 判断对象是否为null,如果为null返回true
System.out.println(Objects.isNull(s1)); // true
System.out.println(Objects.isNull(s2)); // false
```