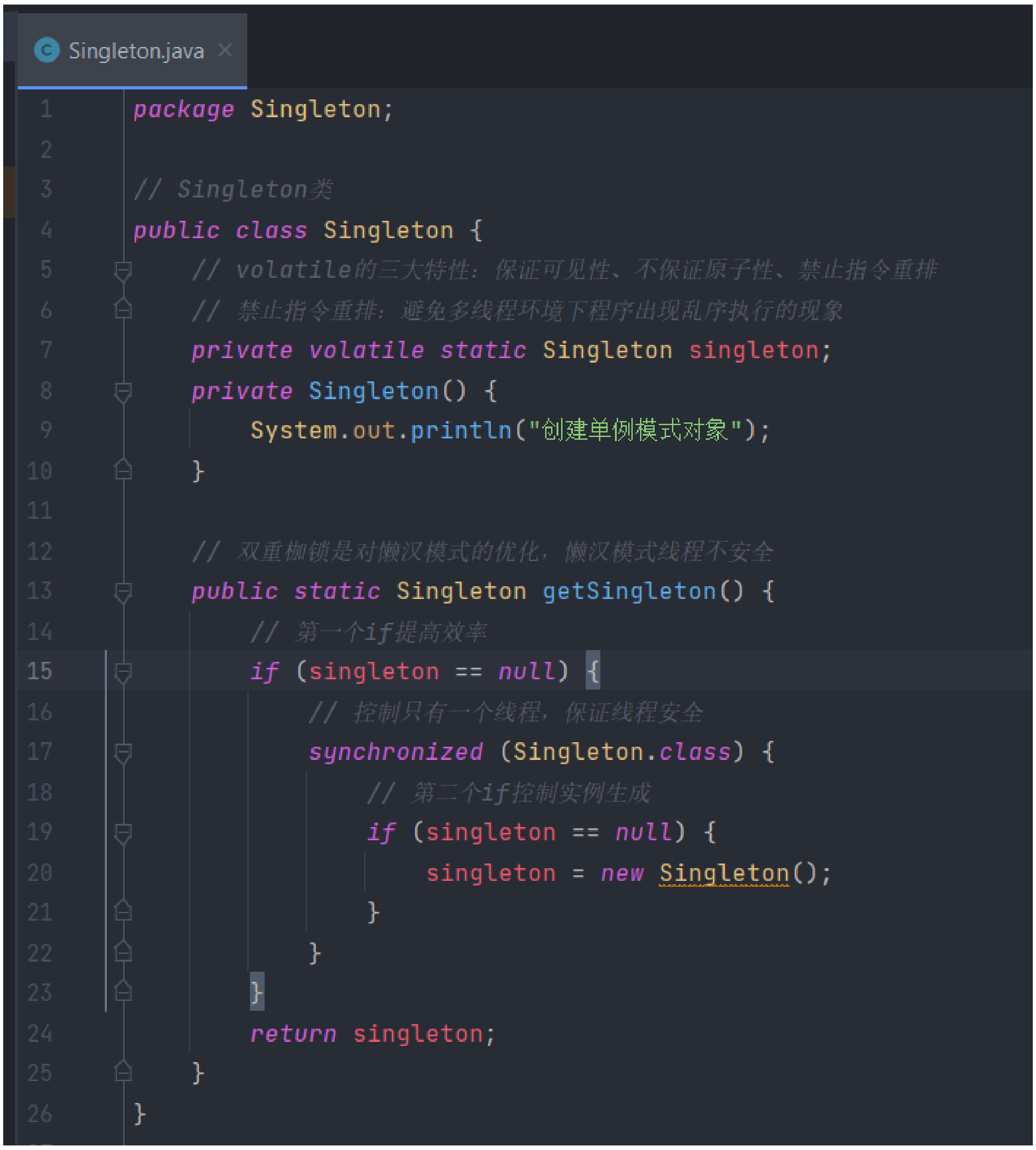
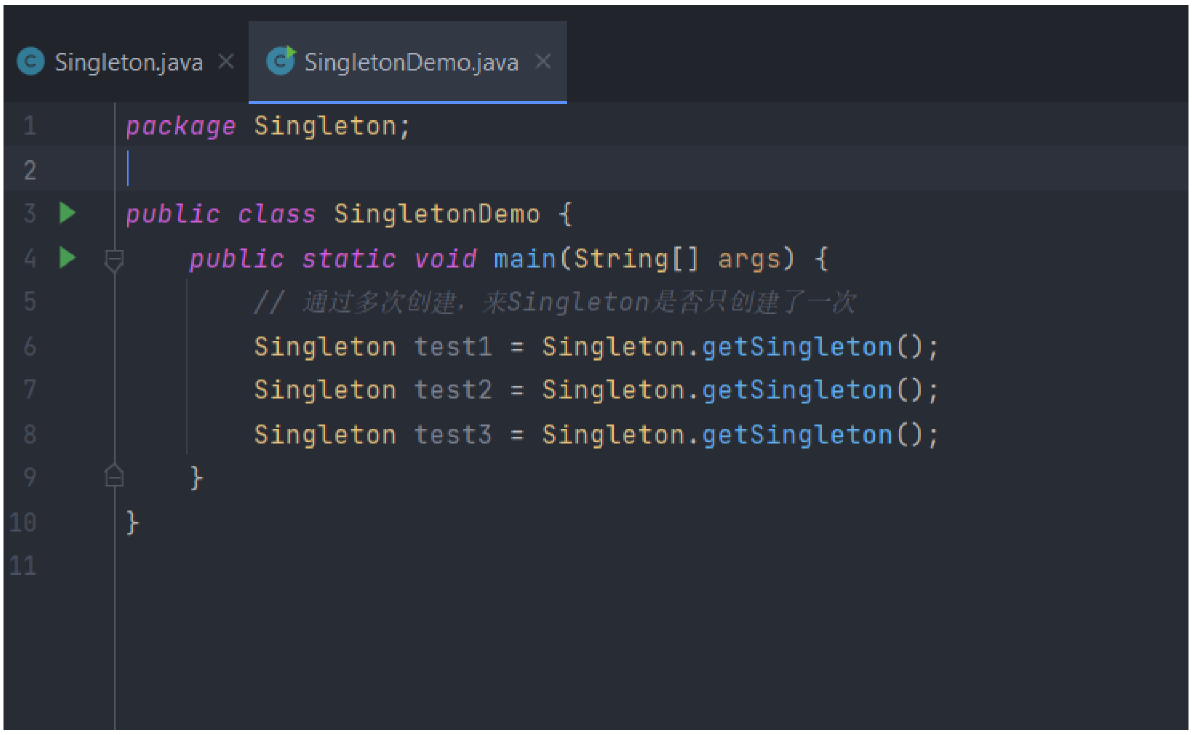
单例模式

代码如下（相关注释写在代码中）:
Singleton 类:
每次实例化 Singleton 类，都会输出“创建单例模式对象”



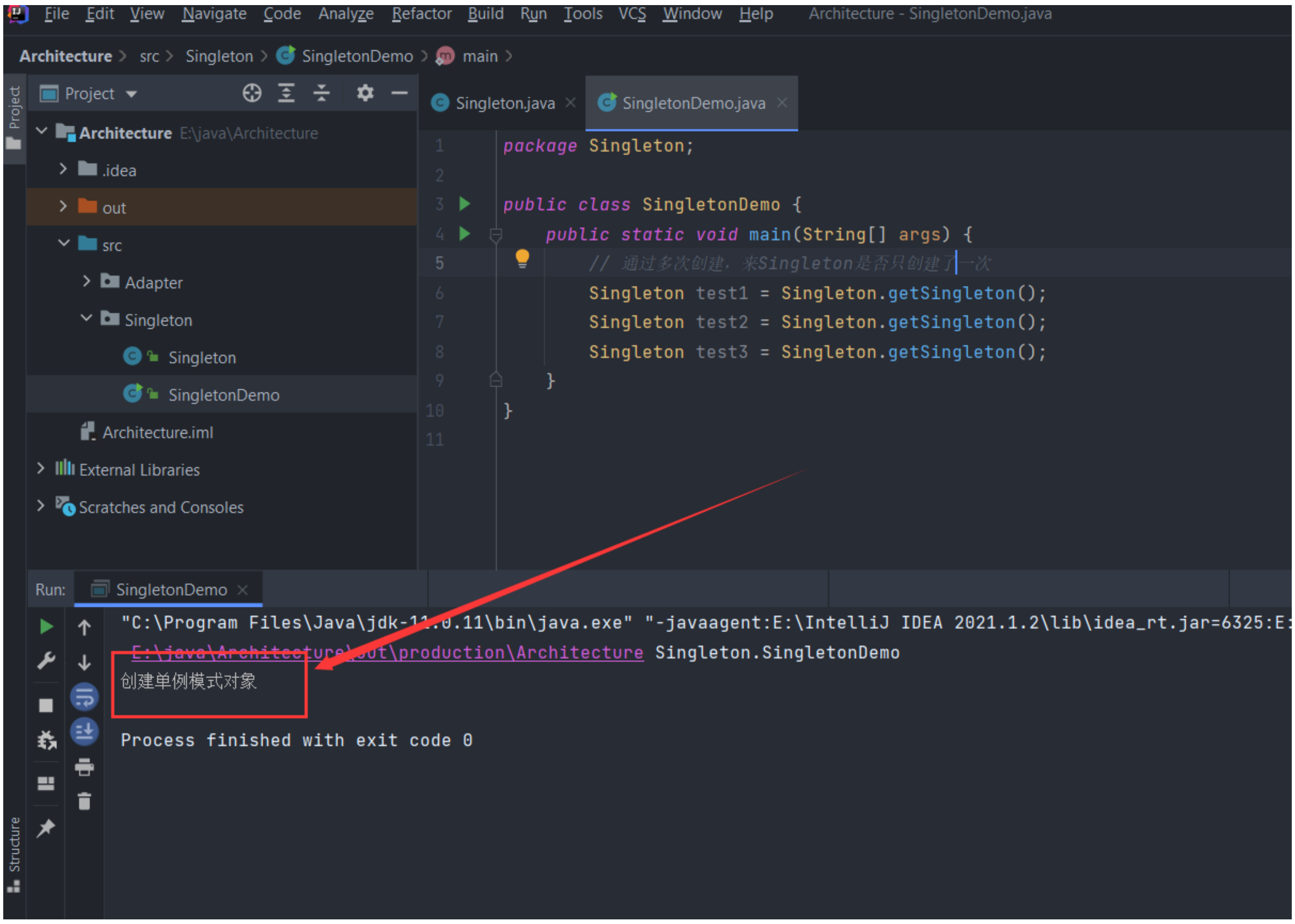
```java
package Singleton;

// Singleton类
public class Singleton {
    // volatile的三大特性：保证可见性、不保证原子性、禁止指令重排
    // 禁止指令重排：避免多线程环境下程序出现乱序执行的现象
    private volatile static Singleton singleton;
    private Singleton() {
        System.out.println("创建单例模式对象");
    }

    // 双重枷锁是对懒汉模式的优化，懒汉模式线程不安全
    public static Singleton getSingleton() {
        // 第一个if提高效率
        if (singleton == null) {
            // 控制只有一个线程，保证线程安全
            synchronized (Singleton.class) {
                // 第二个if控制实例生成
                if (singleton == null) {
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }
}
```

用于测试的 SingletonDemo.java



```java
package Singleton;

public class SingletonDemo {
    public static void main(String[] args) {
        // 通过多次创建，来Singleton是否只创建了一次
        Singleton test1 = Singleton.getSingleton();
        Singleton test2 = Singleton.getSingleton();
        Singleton test3 = Singleton.getSingleton();
    }
}
```

运行结果如下：



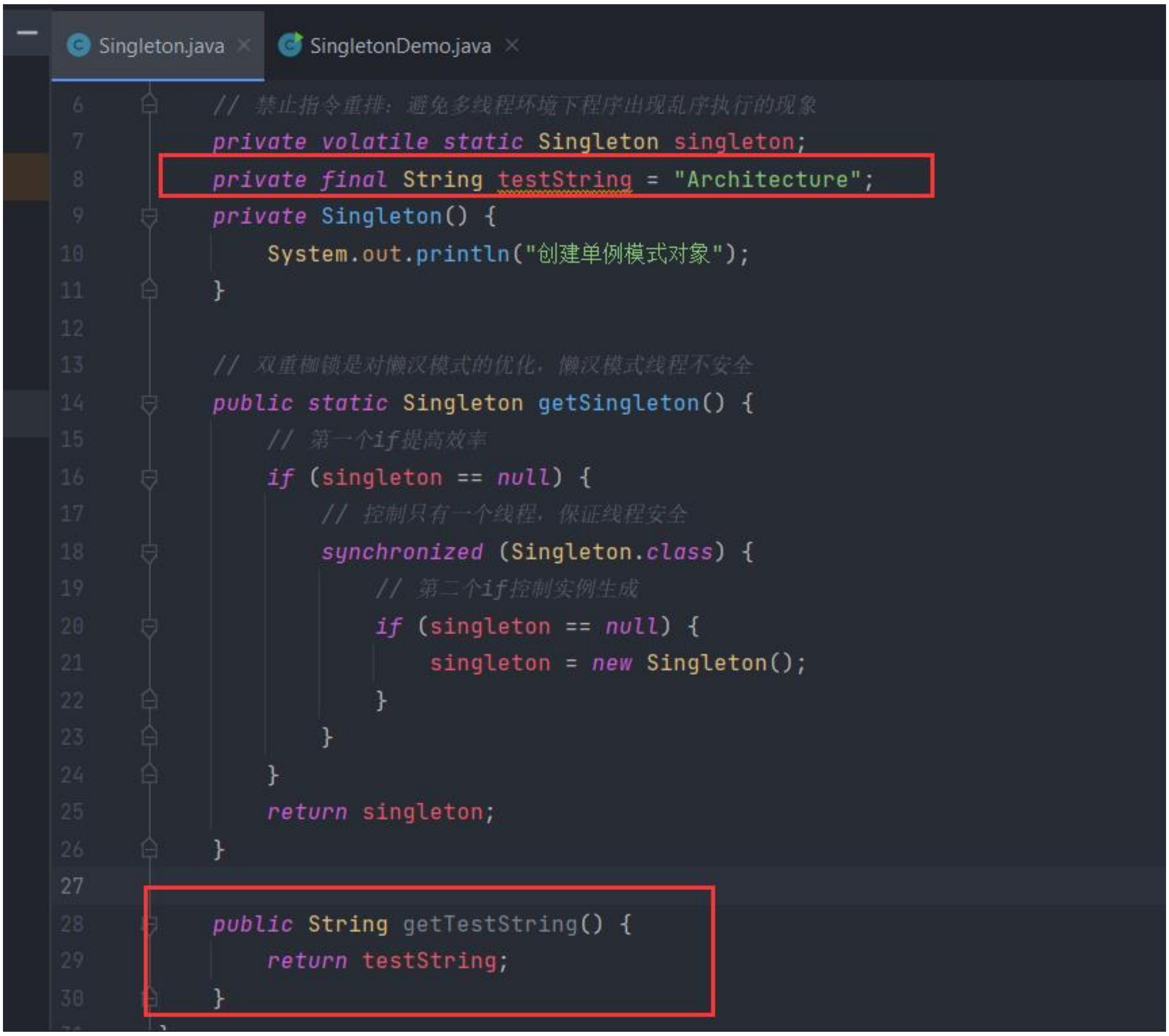
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help Architecture - SingletonDemo.java
Architecture > src > Singleton > SingletonDemo > main
Project
Architecture E:\java\Architecture
.idea
out
src
Adapter
Singleton
Singleton
SingletonDemo
Architecture.iml
External Libraries
Scratches and Consoles
Singleton.java
SingletonDemo.java
Run: SingletonDemo

```java
package Singleton;

public class SingletonDemo {
    public static void main(String[] args) {
        // 通过多次创建，来Singleton是否只创建了一次
        Singleton test1 = Singleton.getSingleton();
        Singleton test2 = Singleton.getSingleton();
        Singleton test3 = Singleton.getSingleton();
    }
}
```

```
"C:\Program Files\Java\jdk-11.0.11\bin\java.exe" "-javaagent:E:\IntelliJ IDEA 2021.1.2\lib\idea_rt.jar=6325:E:
 E:\java\Architecture\out\production\Architecture Singleton.SingletonDemo
创建单例模式对象

Process finished with exit code 0
```

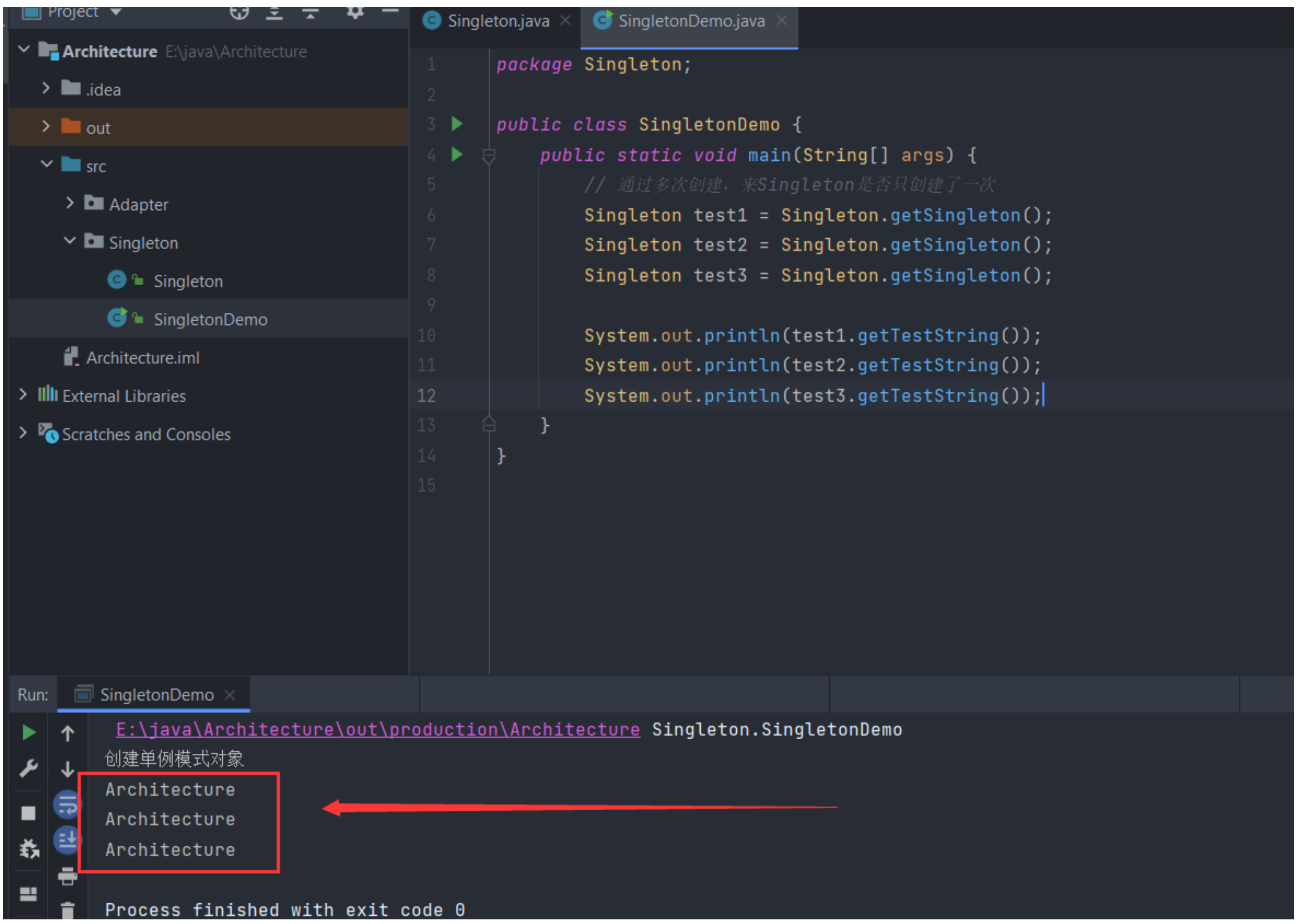
可以看出，虽然我们实例化三次 Singleton 对象，但实际上 Singleton 的构造函数只被执行了一次

我们再为 Singleton 对象添加一个成员变量 testString



```java
    // 禁止指令重排：避免多线程环境下程序出现乱序执行的现象
    private volatile static Singleton singleton;
    private final String testString = "Architecture";
    private Singleton() {
        System.out.println("创建单例模式对象");
    }

    // 双重枷锁是对懒汉模式的优化，懒汉模式线程不安全
    public static Singleton getSingleton() {
        // 第一个if提高效率
        if (singleton == null) {
            // 控制只有一个线程，保证线程安全
            synchronized (Singleton.class) {
                // 第二个if控制实例生成
                if (singleton == null) {
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }

    public String getTestString() {
        return testString;
    }
```

再次运行 SingletonDemo

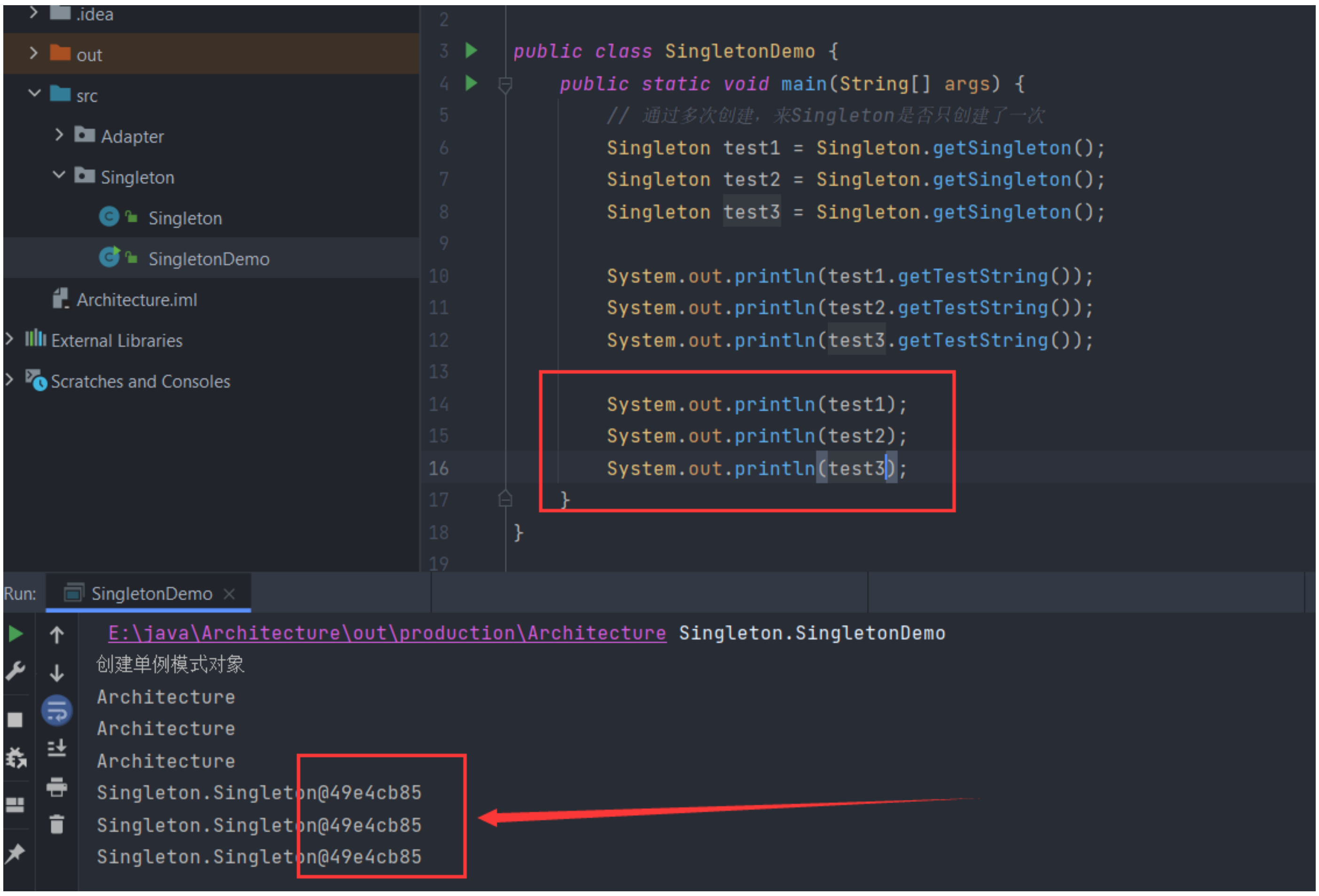


```java
package Singleton;

public class SingletonDemo {
    public static void main(String[] args) {
        // 通过多次创建，来Singleton是否只创建了一次
        Singleton test1 = Singleton.getSingleton();
        Singleton test2 = Singleton.getSingleton();
        Singleton test3 = Singleton.getSingleton();

        System.out.println(test1.getTestString());
        System.out.println(test2.getTestString());
        System.out.println(test3.getTestString());
    }
}
```

```
E:\java\Architecture\out\production\Architecture Singleton.SingletonDemo
创建单例模式对象
Architecture
Architecture
Architecture

Process finished with exit code 0
```

可以看到三个实例化之后的对象的 testString 都是同样的

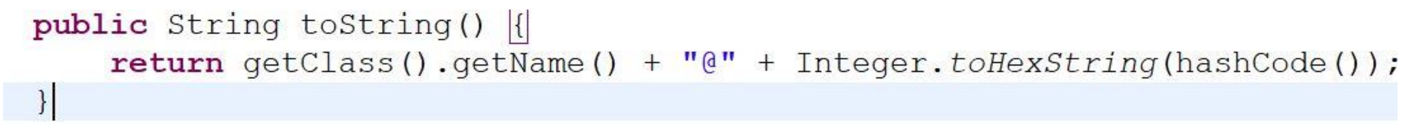
我们再直接输出三个实例化之后的对象



```java
public class SingletonDemo {
    public static void main(String[] args) {
        // 通过多次创建，来Singleton是否只创建了一次
        Singleton test1 = Singleton.getSingleton();
        Singleton test2 = Singleton.getSingleton();
        Singleton test3 = Singleton.getSingleton();

        System.out.println(test1.getTestString());
        System.out.println(test2.getTestString());
        System.out.println(test3.getTestString());

        System.out.println(test1);
        System.out.println(test2);
        System.out.println(test3);
    }
}
```

```
E:\java\Architecture\out\production\Architecture Singleton.SingletonDemo
创建单例模式对象
Architecture
Architecture
Architecture
Singleton.Singleton@49e4cb85
Singleton.Singleton@49e4cb85
Singleton.Singleton@49e4cb85
```

直接输出对象会执行类的 toString 方法，默认的 toString 继承自 Object 类：

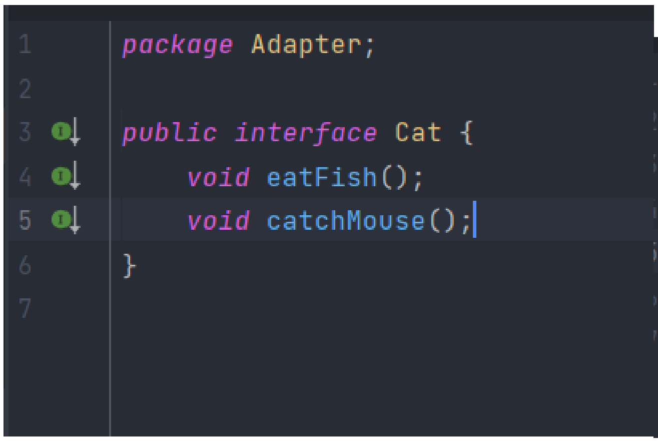


```java
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

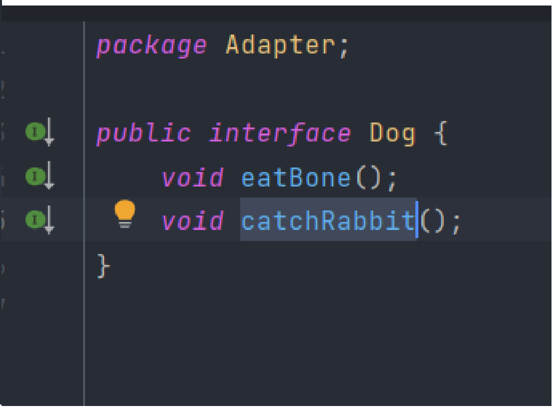
后面的 Integer.toHexString(hashCode())是输出该对象的十六进制内存地址
可以看出，test1 test2 test3 的内存地址都是一样的

适配器

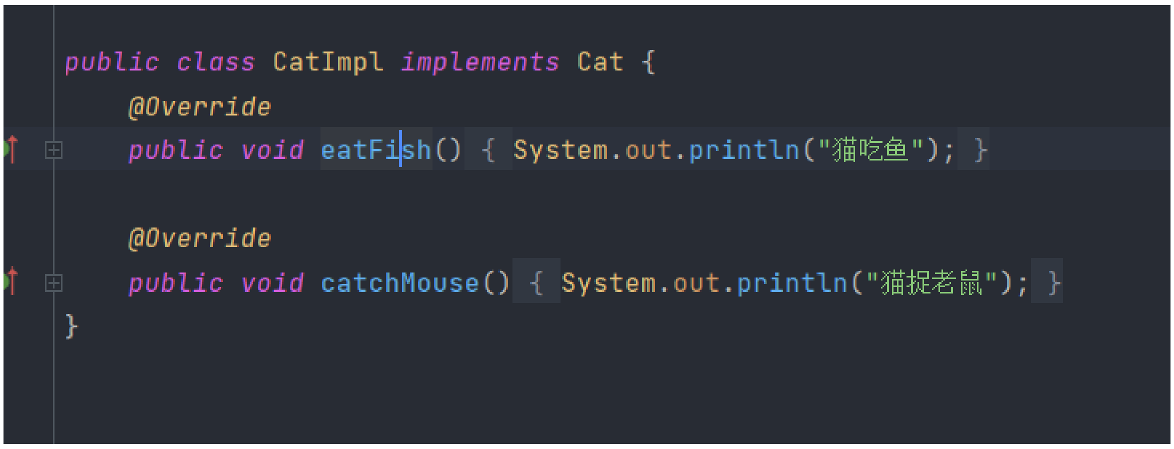
双向适配器，是两个类需要互相适配对方，简单上来说就是双方可以兼容对方的方法
我们有猫狗两个类，猫有抓老鼠、吃鱼方法，狗有抓兔子、吃骨头方法



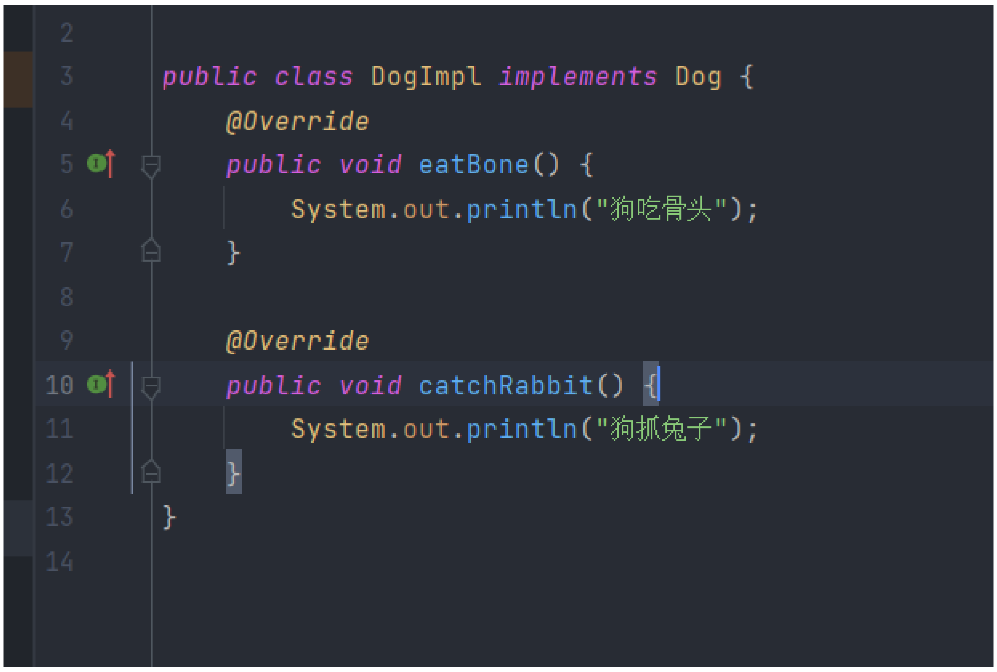
```java
package Adapter;

public interface Cat {
    void eatFish();
    void catchMouse();
}
```



```java
package Adapter;

public interface Dog {
    void eatBone();
    void catchRabbit();
}
```

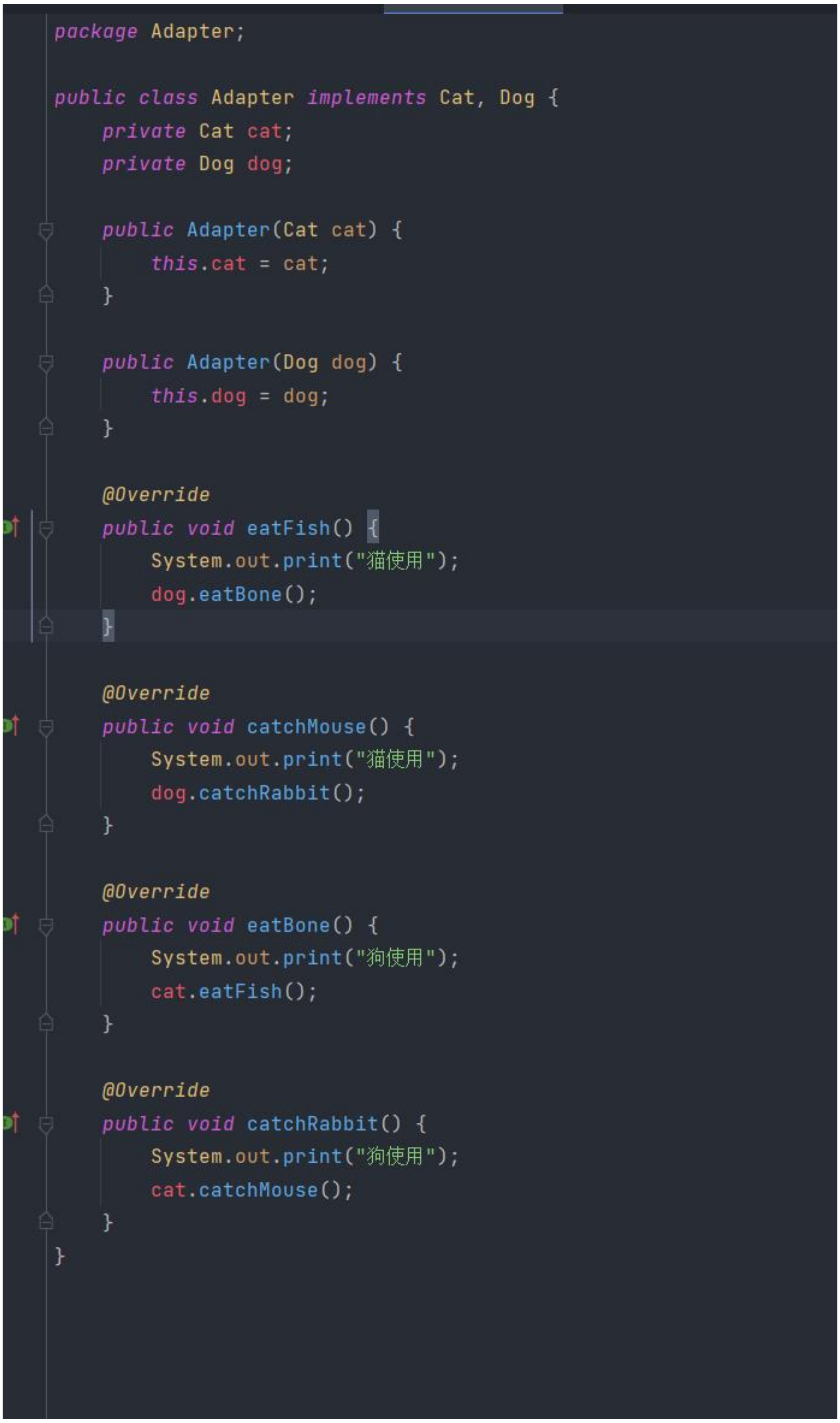


```java
public class CatImpl implements Cat {
    @Override
    public void eatFish() { System.out.println("猫吃鱼"); }

    @Override
    public void catchMouse() { System.out.println("猫捉老鼠"); }
}
```

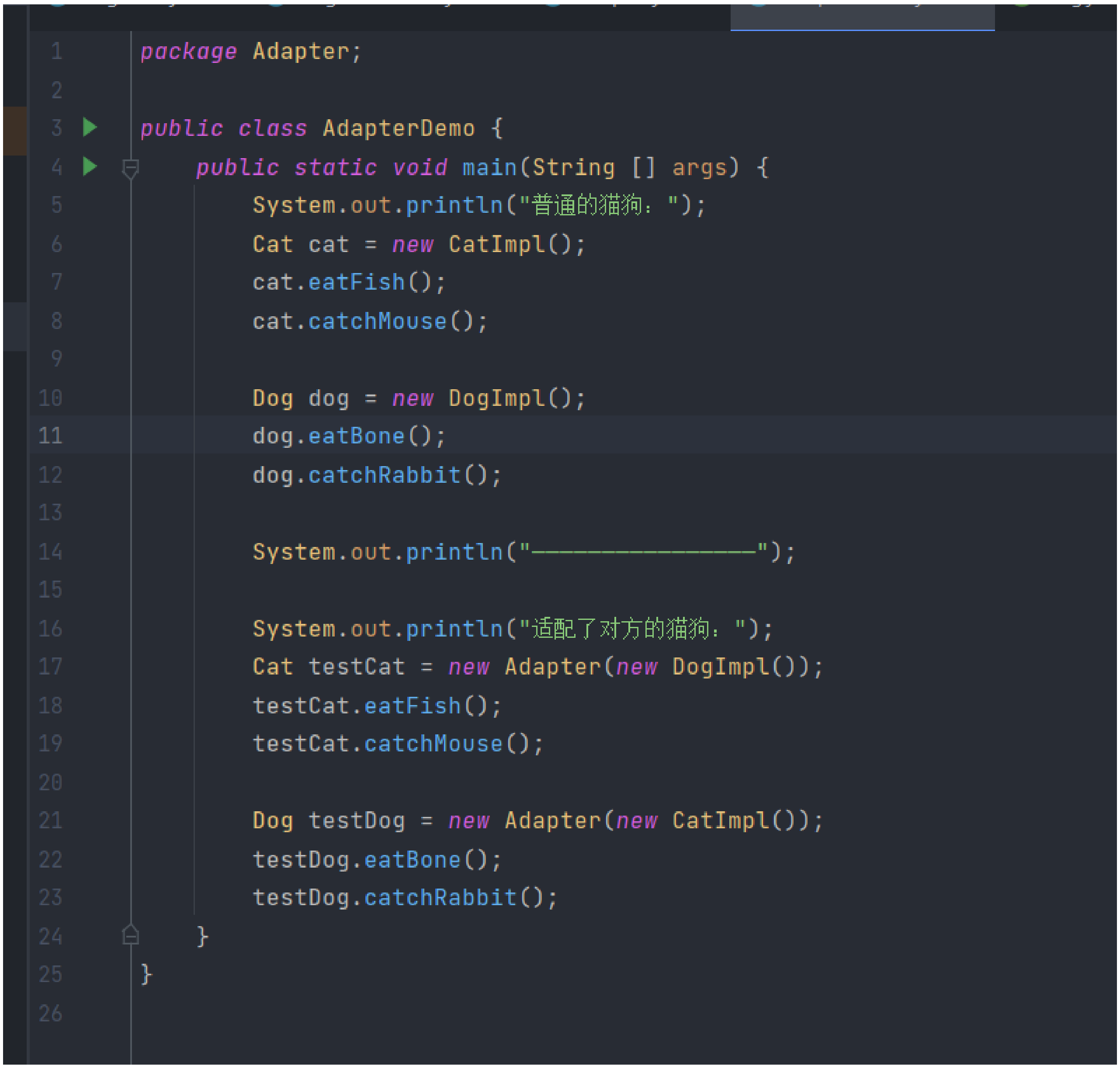


```java

public class DogImpl implements Dog {
    @Override
    public void eatBone() {
        System.out.println("狗吃骨头");
    }

    @Override
    public void catchRabbit() {
        System.out.println("狗抓兔子");
    }
}

```

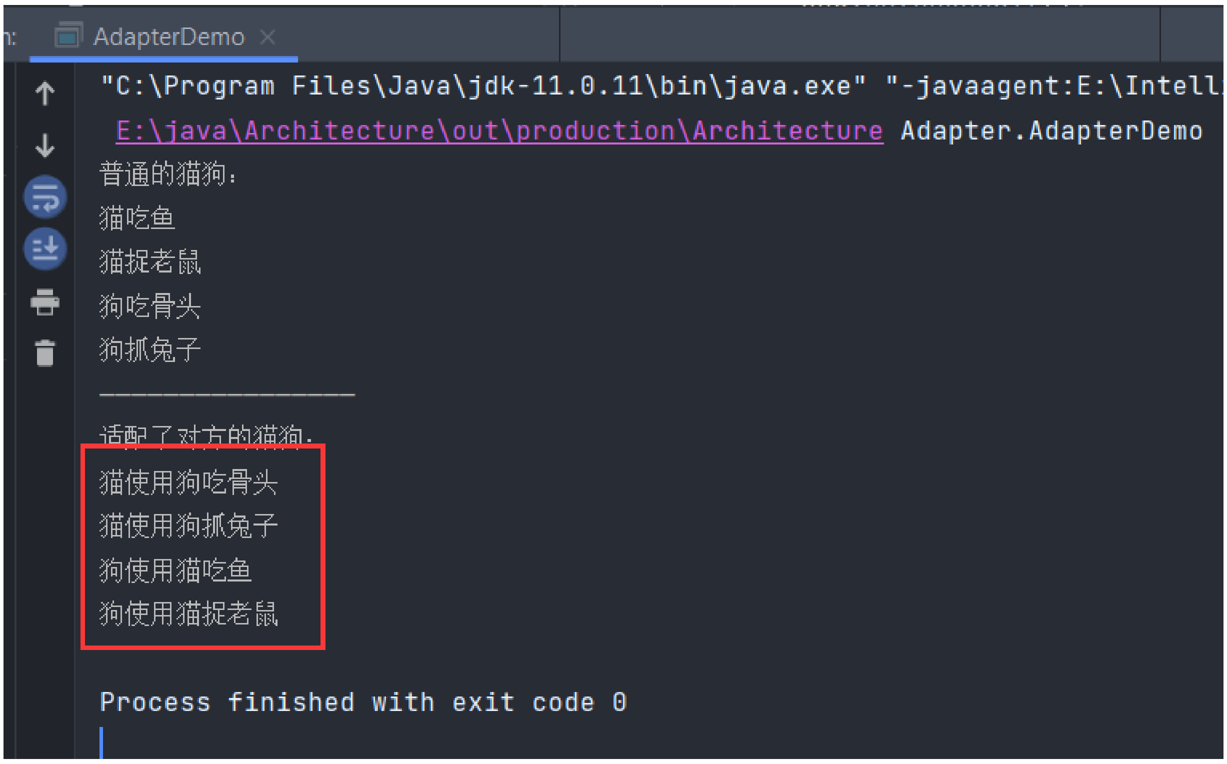
现在我们需要让猫能够抓兔子、吃骨头，狗能吃鱼、捉老鼠。按照一般做法，是直接在对应的类中添加需要的方法。使用适配器可直接实现改变类本身而添加可使用方法：


```java
package Adapter;

public class Adapter implements Cat, Dog {
    private Cat cat;
    private Dog dog;

    public Adapter(Cat cat) {
        this.cat = cat;
    }

    public Adapter(Dog dog) {
        this.dog = dog;
    }

    @Override
    public void eatFish() {
        System.out.print("猫使用");
        dog.eatBone();
    }

    @Override
    public void catchMouse() {
        System.out.print("猫使用");
        dog.catchRabbit();
    }

    @Override
    public void eatBone() {
        System.out.print("狗使用");
        cat.eatFish();
    }

    @Override
    public void catchRabbit() {
        System.out.print("狗使用");
        cat.catchMouse();
    }
}
```

我们简单的写一个类用来测试：



```java
package Adapter;

public class AdapterDemo {
    public static void main(String [] args) {
        System.out.println("普通的猫狗：");
        Cat cat = new CatImpl();
        cat.eatFish();
        cat.catchMouse();

        Dog dog = new DogImpl();
        dog.eatBone();
        dog.catchRabbit();

        System.out.println("————————————————");

        System.out.println("适配了对方的猫狗：");
        Cat testCat = new Adapter(new DogImpl());
        testCat.eatFish();
        testCat.catchMouse();

        Dog testDog = new Adapter(new CatImpl());
        testDog.eatBone();
        testDog.catchRabbit();
    }
}

```



```
"C:\Program Files\Java\jdk-11.0.11\bin\java.exe" "-javaagent:E:\Intell
 E:\java\Architecture\out\production\Architecture Adapter.AdapterDemo
普通的猫狗：
猫吃鱼
猫捉老鼠
狗吃骨头
狗抓兔子
————————————————
适配了对方的猫狗：
猫使用狗吃骨头
猫使用狗抓兔子
狗使用猫吃鱼
狗使用猫捉老鼠

Process finished with exit code 0
```

可以看到，在不改变类本身的情况下，猫狗可以相互使用对方的技能