讲义10：
注解、反射、内部类、Lambda表达式

主要内容

- 讲授内容
  - 注解、反射
  - 内部类
  - 函数式接口
  - Lambda表达式
- 内容资料来源
  - 教材第13章
  - Java API文档

1 注解

- 注解(Annotation)是元数据，元数据是用来描述数据的数据。
- 注解主要用于告诉编译器要做什么事情。
- 注解可以声明在：包、类、接口、成员变量、成员方法、局部变量、方法参数等的前面，对这些程序元素进行注解说明。
- 通过注解，可以在不改变程序逻辑的情况下，在源程序文件中嵌入一些补充信息。然后对过反射机制编程实现对这些注解的访问。

- java.lang.Annotation是注解接口，所有注解都默认实现该接口
- 注解语法：**@注解名(参数表)**

1.1 基本注解

Java 8的java.lang包提供了5个基本注解：

- @Deprecated，表示其注解的程序元素“已经过时，不建议使用”，程序中使用了其注解的程序元素时会出现编译警告。
- @Override，表示其注解的方法必须是对父类方法的覆盖。
- @SuppressWarnings，抑制其注解的程序元素出现警告信息。
- @SafeVarargs，抑制堆污染警告。堆污染是指将一个不带泛型的对象赋值给带泛型的对象。只用于static、final修饰的方法或构造方法，否则编译不通过。
- @FunctionalInterface，表示其注解的接口必须是函数式接口。函数式接口是只有一个抽象方法的接口。

1.1 基本注解

@Deprecated和@Override例子

```
public class Demo1 {
    @Deprecated public void m1() {
        System.out.println("过时的方法");
    }
    @Override public boolean equals(Object obj) {
        return true;
    }
}
```

示例代码：examples/lecture10/annotation_demo的Demo1.java

1.1 基本注解

@SuppressWarnings和@SafeVarargs例子

```
public class Demo2<S> {
    private S[] args;
    @SafeVarargs   public Demo2(S... args) {   this.args = args;   }
    @SuppressWarnings("unchecked")
    public void loopPrintArgs(S... args){
        for (S arg : args) {  System.out.println(arg);  }
    }
    @SafeVarargs  public final void printSelfArgs(S... args){
        for (S arg : this.args) {  System.out.println(arg);     }
    }
    @SafeVarargs  public static <T> void loopPrintInfo(T ... infos){
        for (T info : infos) {   System.out.println(info);    }
    }
}
```

示例代码：examples/lecture10/annotation_demo的Demo2.java

1.1 基本注解

@FunctionalInterface例子

```
@FunctionalInterface
public interface Demo3 {
    void method();
}
```

示例代码：examples/lecture10/annotation_demo的Demo3.java

1.2 元注解

元注解(元数据注解)，是对注解进行标注的注解。

Java的6种元注解（自行了解）：

- @Target
- @Retention
- @Document
- @Inherited
- @Repeatable
- 类型注解

1.3 自定义注解

注解定义的语法格式

自学知识

```
[public] @interface 注解名 {
    数据类型 成员变量名() [default = 初始值]
}
```

2 反射机制

Java对象在运行时可以有2种类型：**编译时类型**和**运行时类型**
例如：Object obj = new String();
obj的编译类型为Object，而执行创建对象后实际对象是String。

Java的反射(reflection)机制是指在程序运行过程中，可以通过对象了解其所属类，获取类的成员变量和方法，调用对象的属性和方法。

Java提供的与反射机制相关的类

- java.lang.Class
- java.lang.reflect.Constructor
- java.lang.reflect.Field
- java.lang.reflect.Method
- java.lang.reflect.Parameter

2 反射机制

获得Class类的对象实例的3种常用方式

```
Circle circle = new Circle(10.0);

Class c1 = reflect_demo.Circle.class;
Class c2 = Class.forName("reflect_demo.Circle");
Class c3 = circle.getClass();
```

示例代码：examples/lecture10/reflect_demo

2 反射机制

Class类及reflect包类的使用示例

```
Class c = ...;  // Class类的对象

String className = c.getName());
Package p = c.getPackage();
Class superClass = c.getSuperclass();
Class[] interfaces = c.getInterfaces();
Constructor[] constructors = c.getDeclaredConstructors();
Field[] fields = c.getDeclaredFields();
Method[] methods = c.getDeclaredMethods();
```

示例代码：examples/lecture10/reflect_demo

3 内部类

内部类是定义在其他类中的类。

匿名内部类是一种特殊的内部类，没有类名。

3.1 内部类

Java允许在类的内部定义类，称为**嵌套类(Nested Class)**。

嵌套类分为2种：

- 静态嵌套类：使用static修饰
- 非静态嵌套类：也称为内部类(Inner Class)

```
class OuterClass {
  static class StaticNestedClass {
  }
  class InnerClass {
  }
}
```

编译的目标文件：
OuterClass.class
OuterClass$InnerClass.class
OuterClass$StaticNestedClass.class

3.1 内部类

① 一个嵌套类是其外部类的一个成员，其访问权限可以声明为public、private、protected或包。

② 内部类(非静态嵌套类)中可以直接访问其外部类的其他成员，即使这些成员使用private修饰。
当内部类与其外部类有同名实例成员时，如成员名data
- 内部类的成员：this.data
- 外部类的成员：外部类名.this.data

③ 静态嵌套类中不能访问其外部类的实例成员；但是可以访问其外部类的静态成员。

④ 外部类中可以访问其嵌套类的所有成员，即使这些成员使用private修饰。

示例：examples\Lecture10\innerclass_demo

3.1 内部类

Why Use Nested Classes?

- **根本原因**：这是一种对只在一个地方使用的类进行逻辑分组的方式。
  如果类A只在类B中使用, 则A应该成为B的嵌套类。
- 这种方式提升了封装。
  如果类A是类B的嵌套类, 就可以使用更多的访问权限去封装的控制B的使用。
- 这种方式可以使代码的可读性和可维护性更高。

PS: 独立定义的类称为**顶层类(Top-Level Class)**，访问权限只有public 或 包 2种。

3.1 内部类

尽可能只在外部类中使用其嵌套类，但是如果访问权限允许，语法允许在其它位置创建嵌套类对象。

创建静态嵌套类对象的例子：

```
OuterClass.StaticNestedClass  nestedObject =
                    new OuterClass.StaticNestedClass();
```

创建内部类对象的例子：

```
OuterClass  outerObject = new OuterClass();
OuterClass.InnerClass innerObject =
                    outerObject.new InnerClass();
```

3.2 匿名内部类

- 匿名类允许同时定义和实例化一个类。
- 匿名类实际上是一个没有名字的内部类。

匿名类表达式由以下几个部分组成：

- new
- 匿名类需要实现的接口名或需要继承的父类名
- 圆括号，其中包含需要传递给(父类)构造方法的参数
  如果匿名类实现的是接口，参数表保持为空。
- 类的主体，{}包围，其中可以声明数据域或方法。

示例：examples\Lecture10\anonymous_class_demo

3.2 匿名内部类

使用继承父类方式创建匿名类及对象的示例

```
public abstract class Greeter {
   protected String name;
   public Greeter(String name) {this.name = name;}
   public abstract void sayHi();
}
```

```
Greeter first = new Greeter("Tom") {
   @Override
   public void sayHi() {
      System.out.println("Hello " + name);
   }
};
first.sayHi();
```

3.2 匿名内部类

使用实现接口方式创建匿名类及对象的示例

```
public interface Greeting {
   public void greet();
}
```

```
Greeting second = new Greeting() {
   @Override
   public void greet() {
      System.out.println("Hello World!");
   }
};
second.greet();
```

4 函数式接口与Lambda表达式

函数式接口与Lambda表达式是JDK 8开始引入的概念。

所谓的函数式接口，首先是一个接口，然后就是在这个接口里面只能有一个抽象方法。也称为SAM接口，即Single Abstract Method interfaces。

Lambda表达式是在应用函数式接口环境下一种简化定义形式，可以解决匿名内部类定义形式复杂的缺点。

4.1 函数式接口

一个函数式接口如下：

```
@FunctionalInterface
public interface GreetingService  {
    void sayMessage(String message);
}
```

函数式接口中允许增加3种方法：

```
@FunctionalInterface
public interface GreetingService {
    void sayMessage(String message);
    boolean equals(Object obj); //Object类的public方法
    default void doSomeMoreWork1() { //...  }
    static void doSomeMoreWork1() { //...  }
}
```

4.2 Lambda表达式

lambda表达式的一般语法：

```
(Type1 p1, Type2 p2, ..., TypeN pN) -> {
   statment1;
   statment2;
   //.............
   return statmentM;
}
```

单参数语法可以简化为：

```
p1 -> {
   statment1;
   statment2;
   //.............
   return statmentM;
}
```

单语句语法简化为：

```
param1 -> statment
```

4.2 Lambda表达式

```
//传统匿名内部类的实现
Greeting first = new Greeting() {
    @Override
    public void greet() {
        System.out.println("Hello World!");
    }
};
first.greet();

// Lambda表达式的实现
Greeting second = () -> {
    System.out.println("Hello Lambda!");
};
second.greet();
```

示例：examples\Lecture10\lambda_demo

4.3 Lambda表达式作为方法的参数

Lambda表达式可以作为实参传递给方法。方法对应的形参必须是一个函数式接口类型。

java.util.function包中定义了大量函数式接口，方便开发过程中使用Lambda表达式。

以 java.util.function.Function<T, R> 为例，其源码如下：

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
    //省略default和static方法
}
```

示例：examples\Lecture10\lambda_para_demo

4.3 Lambda表达式作为方法的参数

定义一个静态方法process，其2个参数：

- Function<String, String> func，接口的抽象方法处理src
- String src，被处理的字符串

返回值：处理后的结果String

```
public static String process(
                Function<String, String> func,
                String src)
{
    return func.apply(src);  //调用接口的方法处理src
}
```

4.3 Lambda表达式作为方法的参数

调用示例：

```
String str = "Hello Lambda Expression";
// 使用Lambda表达式传入String对象的toUpperCase方法
String out1 = process((s) -> s.toUpperCase(), str);

// 使用Lambda表达式传入一个自定义方法
String out2 = process((s) -> {
    String result = "";
    for (int i = s.length() - 1; i >= 0; i--) {
        result += s.charAt(i);
    }
    return result;
}, str);
```

4.3 Lambda表达式作为方法的参数

```
// 使用传统匿名内部类方式也是可以的
String out3 = process(new Function<String, String>() {
    @Override
    public String apply(String s) {
        String result = "";
        for (int i=0; i<s.length(); i+=2) {
            result += s.charAt(i);
        }
        return result;
    }
}, str);
```

5 方法引用

JDK 8引入“::”运算符用于方法引用，可以简化Lambda表达式仅仅调用一个已经存在的方法的情况。方法引用的4种形式：

- 特定实例对象的方法引用，对象名::实例方法名
- 静态方法引用，类名::静态方法名
- 任意对象(同一个类)的实例方法引用，类名::实例方法名
- 构造方法引用，类名::new

说明：

- 被引用方法的参数列表和返回值必须与函数式接口中的抽象方法一致。
- 引用方法时，::后面只能是方法名或new，没有()。
- 前两种方式用于只有一个参数情况；第3种方式允许多个参数，第1个参数是调用方法的对象。
- 当引用重载方法时，JVM自动判定并调用合适的方法。

5 方法引用

示例1：对象名::实例方法名 演示

本例基于函数式接口java.util.function.Consumer<T>演示方法引用的应用场景。该接口的结构如下：

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
    // 省略default方法
}
```

示例：examples\Lecture10\method_ref_demo 中的Demo1.java

5 方法引用

示例1：对象名::实例方法名 演示

第1种实现：Consumer的匿名内部类对象并调用accept方法

```
Consumer<Object> con1 = new Consumer<Object>() {
    @Override
    public void accept(Object obj) {
        System.out.println(obj.toString());
    }
};
con1.accept(new Date());
```

方法体内只有一条对System.out对象println方法的调用

第2种实现：Lambda表达式实现

```
Consumer<Object> con2 = obj -> System.out.println(obj);
con2.accept(new Date());
```

5 方法引用

示例1：对象名::实例方法名 演示

本例实现函数式接口的抽象方法时，只有一条对某个确定对象的方法的调用语句，这种情况下可以使用“对象名::实例方法名”的方法引用简化代码。

```
Consumer<Object> con3 = System.out::println;
con3.accept("方法引用的实现消费型接口！ ");
```

5 方法引用

示例2：类名::静态方法名 演示

本例基于函数式接口java.util.function.Function<T>和静态方法引用实现数值与字符串之间的转换。

```
Function<String, Integer> s2i = Integer::parseInt;
Function<String, Double> s2d = Double::parseDouble;
Function<Number, String> n2s = String::valueOf;

int k = s2i.apply("123");
double d = s2d.apply("3.14");
String s = n2s.apply(123.F);
```

示例：examples\Lecture10\method_ref_demo 中的Demo2.java

5 方法引用

示例3：类名::实例方法名 演示

本例基于函数式接口和实例方法引用的字符串处理。

(1) 方法引用实现 "Hello Java".toLowerCase()

被引用方法没有形参，需要函数式接口的抽象方法有一个参数。

```
Function<String, String> lower = String::toLowerCase;
String out1 = lower.apply("Hello Java!");
```

"Hello Java".toLowerCase()

示例：examples\Lecture10\method_ref_demo 中的Demo3.java

5 方法引用

示例3：类名::实例方法名 演示

(2) 方法引用实现 "Hello Java!".concat("2019")

被引用方法有1个形参, 需要函数式接口的抽象方法有2个参数。

```
BiFunction<String, String, String> concat = String::concat;
String out3 = concat.apply("Hello Java!", "2019");
```

第1个参数的传递

第2个参数的传递

```
"Hello Java!".concat("2019")
```

5 方法引用

示例4：类名::new 演示

```
public class Person {
    public Person(String name, LocalDate birthday) {
        this.name = name;
        this.birthday = birthday;
    }
    // ......
}
```

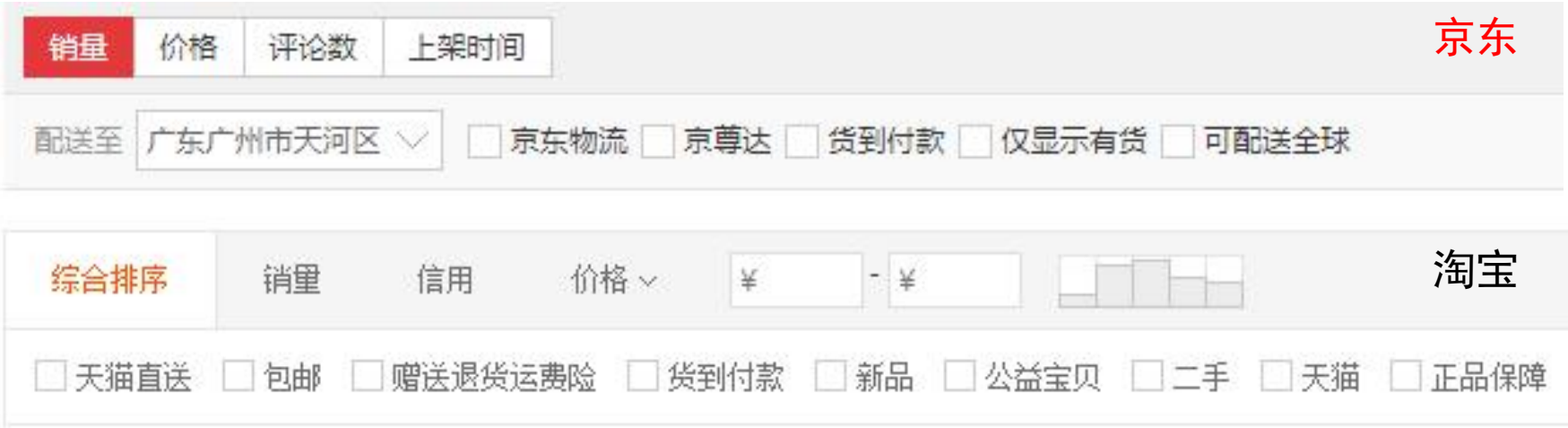
```
@FunctionalInterface
public interface BiParameterCreator<T, P1, P2> {
    T create(P1 p1, P2 p2);
}
```

```
BiParameterCreator<Person, String, LocalDate> creator = Person::new;
Person p1 = creator.create("John", LocalDate.of(2000, 2, 20));
```

示例：examples\Lecture10\method_ref_demo 中的Demo4.java

6 综合应用示例

目标：对一个对象按其多个不同的属性进行排序，这个问题的应用场景非常广泛，例如：



示例：examples\Lecture10\productList

其中演示了各种实现方法，开发时选择一个自己习惯的方式。

课后工作

- 结合教材和Java API文档，要求掌握：
  - 泛型的定义与使用
  - 容器类的概念与常用的线性表、集合、映射的使用
- 自行练习教材的上机实践和习题