







四、五、六

2018年12月21日 19:05

第四章

- 类之间关系：UML（统一建模语言）

表 4-1 表达类关系的 UML 符号

关 系	UML 连接符
继承	
接口实现	
依赖	
聚合	
关联	
直接关联	

- 纪元 (epoch)：时间表示为举例一个固定时间点的毫秒数（可正可负），这个固定时间点就为纪元。是UTC时间1970年1月1日 00:00:00
- 访问器方法，更改器方法：访问类的实例但不更改的方法，能更改类的实例数据的方法。也叫域访问器方法和域更改器方法。
- LocalDate：java内置日历类（~~~Calendar又不用了？），将时间和日历区分。使用静态工程方法构造：LocalDate.now(), LocalDate.of(2001, 12, 12)。LocalDate和Calendar的区别是：前者使用plusDays(int)结果返回一个新的对象，原对象不变；后者add(Calendar.DAY_OF_MONTH, int)是改变原对象
- java编译器编译一个类X时，遇到使用了其他自定义类Y时，会去查找Y.class文件，如果没有找到，则会查找Y.java，然后编译。如果查到Y.class了，但Y.class版本比Y.java版本旧，也会自动重新编译Y.java为Y.class
- NumberFormat：
 - NumberFormat.getCurrencyInstance() // 获取当前地区的货币表示格式
 - NumberFormat.getPercentInstance() // 获取百分数表示格式
- Java是一种按值调用的语言
- 初始化块：{}，在实例化对象的时候这些块就会被执行，实例化对象时类变量初始化为默认值，父类构造器，类变量定义时初始化语句和初始化块（按顺序），构造器初始化，顺序优先级递减

第五章

- 子类覆盖父类方法后，使用super.targetMethod调用父类原方法
- 多态：一个对象变量可以指示多种实际类型的现象被称为多态
- 方法的名字和参数列表称文方法的签名，返回类型不是签名的一部分，但是在覆盖方法时，允许将返回类型定义为原返回类型的子类。
- 方法重写和覆盖时的如何确定调用哪个方法：(假设有父类A，方法A.test();有子类B，方法B.test();)
 - 编译器查看对象的声明类型和方法名。假设调用x.f(param), 且隐式参数x声明为C类对象。需要注意可能存在多个名字为f的方法，编译器为——列举C类中名为f的方法和其超类中访问属性为public且名为f的方法。
 - 编译器查看调用方法时提供的参数类型，如果参数完全匹配则选择匹配方法，或者去找类型转换后匹配的，这个过程被称为**重载 (overloading) 解析**。如果没找到匹配的则报错
 - 如果是private、static、final方法或构造器，则编译器已经可以准确知道该调用哪个方法，这种调用方式称为**静态绑定**
 - 对应地，调用方法依赖于隐式参数的实际类型，并且在运行时**动态绑定**，虚拟机调用x所引用对象的实际类型最合适的那个类的方法（即如果x的实际类型D中有则调用，否则查找D的父类）
 - 实际虚拟机机会为每个类创建一个方法表，列出所有方法的签名和实际调用的方法，减少每次方法调用的搜索时间。（调用super.f(param), 实际会搜索父类的方法表）
- 编写完美equals方法的建议：
 - 显式参数命名为otherObject，稍后将它转换为other的变量
 - 如果是在子类中重新定义equals，则调用super.equals(otherObject)
 - 检测this与otherObject是否引用同一个对象：if(this==otherObject) return true ;
 - 检测otherObject是否为null，如果为null，返回false
 - 比较this和otherObject是否属于同一个类
 - 如果equals的语义在每个子类中有所改变则使用getClass检测：if(getClass() != otherObject.getClass()) return false ;
 - 如果equals所有子类都拥有同一的语义，就是用instanceof检测：if(!(otherObject instanceof ClassName)) return false ;
 - 将otherObject转换为相应的类型：ClassName other = (ClassName)otherObject;
 - 现在开始比较域值，使用==比较基本数据类型，使用Objects.equals比较对象域。
- 自动装箱规范要求：boolean、byte、char<=127, 介于-128~127之间的short和int被包装到固定的对象中，所有他们的结果一定相等
- 枚举类型：ordinal方法返回enum声明中枚举常量的位置，位置从0开始：Size.SMALL.ordinal()
- 反射**：能够分析类能力的程序称为反射
- 异常（包括Exception和Error）：分为**已检查异常**（checked exceptions）和**非检查异常**（unchecked exceptions）。其中只针对Exception异常可以划分为**运行时异常**（RuntimeException及其子类）和**非运行时异常**。非检查异常包括运行时异常和Error。

第六章

- 接口中不能有实例域和静态方法，方法默认为public，接口中的域自动设为常量public static final
- 在Java8中，允许在接口中增加静态方法（这是为了取消伴随工具类，比如Collection的伴随工具类Collections中的方法可以使用静态方法实现，从而不再需要伴随工具类）；可以为接口方法提供一个默认实现，必须用default修饰符标记（减少不必要的方法实现，避免接口演化，即接口方法增加时，原先的实现接口的类不需要更新）。
- 默认方法冲突问题：
 - 超类优先：如果超类提供一个具体方法，同名且有相同参数的默认方法会被忽略
 - 接口冲突：如果一个超接口提供了一个默认方法，另一个接口提供了同名且相同参数的方法（不管是否有默认实现），必须覆盖整个方法来解决冲突。如果两个接口都没用默认实现，则和java8之前一样，该类必须实现或定义为抽象类
- 对象的clone：
 - 对象实现Cloneable接口（这是一个标记接口，只是表示这个对象能被clone，如果不实现这个接口则会抛出CloneNotSupportedException）
 - 重写clone方法，定义为public
 - 调用super.clone()
- lambda表达式-函数式接口：对于只有一个抽象方法的接口，需要这种接口的对象时，就可以提供一个lambda表达式。（java.util.function包中定义了很多通用的函数式接口）

表 6-1 常用函数式接口

函数式接口	参数类型	返回类型	抽象方法名	描述	其他方法
Runnable	无	void	run	作为无参数或返回值的动作运行	
Supplier<T>	无	T	get	提供一个 T 类型的值	
Consumer<T>	T	void	accept	处理一个 T 类型的值	andThen
BiConsumer<T, U>	T, U	void	accept	处理 T 和 U 类型的值	andThen
Function<T, R>	T	R	apply	有一个 T 类型参数的函数	compose, andThen, identity
BiFunction<T, U, R>	T, U	R	apply	有 T 和 U 类型参数的函数	andThen
UnaryOperator<T>	T	T	apply	类型 T 上的一元操作符	compose, andThen, identity
BinaryOperator<T>	T, T	T	apply	类型 T 上的二元操作符	andThen, maxBy, minBy
Predicate<T>	T	boolean	test	布尔值函数	and, or, negate, isEqual
BiPredicate<T, U>	T, U	boolean	test	有两个参数的布尔值函数	and, or, negate

- lambda表达式结构：
 - 一个代码块
 - 参数
 - 自由变量的值，指非参数且不是代码块中定义的变量（这个自由变量是lambda表达式所在嵌套块里的变量，而且变量是最终变量，该变量在外部作用域里不可变，在内部也不可改变。）
- 方法引用：使用现成方法完成想要传递到其他代码的动作。（可以使用this和super，this::instanceMethod）
 - object::instanceMethod
 - Class::staticMethod
 - Class::instanceMethod（第一个参数会成为方法的目标）
- 构造器引用：ClassName::new, (会根据上下文自动判断选择哪个构造器)
- 变量作用域：lambda表达式代码块与表达式所在的嵌套块有相同的作用域（所以不能有同名的局部变量，this指的是外围的类）
- 需要内部类的原因：
 - 内部类可以访问该类定义所在作用域中的数据，包括私有数据
 - 内部类可以对同一个包中的其他类隐藏起来
 - 定义回调函数时，匿名内部类比较便捷
- 内部类的机制：内部类的对象含有一个隐式引用指向创建了它的外部类对象，这个引用在内部类的定义中不可见，是编译器修改了内部类的构造器，添加了一个外围类的引用参数,这个引用正规语法为OuterClass.this
- **内部类中声明的静态域必须是final，内部类不能有static方法**
- 内部类能够访问外围类私有变量的原因：编译器在外围类添加了访问私有变量的包可见的静态方法access\$0(这个方法名不固定，和编译器有关），这个方法返回私有变量。这个方法名不是合法的java语法规则，所以不会被用户代码调用（但是手动创建class文件就可以）
- 虚拟机不存在私有类
- 局部内部类：定义在外围类方法体里的类
 - 局部类不能用public或private修饰
 - 局部类被完全隐藏起来，外围类其他代码也不能访问它
 - 局部类可以访问外围类的局部变量（访问的实质是将局部变量作为构造参数传递给内部类，成为内部类的final static域，所以不可变）
- 静态内部类：只有内部类可以声明为static，
 - 静态内部类没有对生成它的外围类对象的应用特权外。即静态内部类不含有外围类对象的引用，所以静态内部类的创建往往在静态方法中。
 - 静态内部类可以有静态语和方法
 - 声明在接口中的内部类自动成为static和public
- 调用处理器:实现InvocationHandler接口，含方法：Object invoke(Object proxy,Method method,Object[]args);
- 代理类机制：
 - 代理类在程序运行时创建，只含有一个实例域，调用处理器，所以附加的数据都必须存储在调用处理器中
 - 调用代理类的方法时，实质是调用了该代理类中的调用处理器对象的invoke方法，而该方法执行完用户定义逻辑语句后执行被代理类target的同名方法。
- 代理类将会代理：
 - 初始化Proxy.netProxyInstance(ClassLoader,Class[], InvocationHandler)方法中的Class数组参数里的接口
 - Object的一些方法： toString, equals, hashCode
- 对于特定的类加载器和预设的一组接口，只能有一个代理类，如果使用两次相同的加载器和接口数组调用newProxyInstatnce方法，只是得到同一个类的两个对象。