10.抽象数据类型

1. **客户端对类型内部表示做出假设的危险**
   - **示例**：假设有一个 `Wallet` 类来表示钱包，如下：

java

```
public class Wallet {
    private int amount = 0;
    public void loanTo(Wallet that) {
        that.amount += this.amount;
        this.amount = 0;
    }
}
```

- **危险分析**：如果客户端代码像这样编写：
  java

```
public class Person {
    private Wallet w;
    public int getNetWorth() {
        return w.amount;
    }
    public boolean isBroke() {
        // 这里客户端错误地假设可以直接访问Wallet的amount字段，违反封装原则
        return Wallet.amount == 0;
    }
}
```

- **危险后果**：在上述 `Person` 类的 `isBroke` 方法中，客户端直接访问 `Wallet` 类的 `amount` 字段，假设了 `amount` 字段的存在和可访问性。如果 `Wallet` 类的内部表示发生变化，例如将 `amount` 字段重命名，或者改变其访问修饰符， `Person` 类的这个方法就会出错，导致程序出现难以调试的错误。这体现了客户端对类型内部表示做出假设的危险性。

1. **避免该危险的方法**：通过抽象数据类型，将数据的表示和操作封装起来。例如， `Wallet` 类只提供像 `loanTo` 这样的公共方法来操作 `amount` 字段，客户端只能通过这些公共方法与 `Wallet` 对象交互。即使 `Wallet` 类的内部表示改变，只要公共方法的接口和功能不变，客户端代码无需修改。

2. **操作的分类**
   - **创建者（Creator）**：创建该类型的新对象。例如，自定义一个 `MyString` 类（简化版模拟字符串功能），构造函数就是创建者操作。

收起

java

```
public class MyString {
    private char[] chars;
    public MyString(String s) {
        chars = s.toCharArray();
    }
}
```

- **生产者（Producer）**：采用该类型的一个或多个现有对象作为输入，生成新对象。例如，`String` 类的 `concat` 方法，接受另一个字符串，生成新的连接后的字符串。

收起

java

```
String str1 = "Hello";
String str2 = " World";
String newStr = str1.concat(str2);
```

- **观察者（Observer）**：采用抽象类型的对象并返回不同类型的对象。例如，`String` 类的 `length` 方法，返回字符串的长度（`int` 类型）。

收起

java

```
String str = "example";
int length = str.length();
```

- **变更者（Mutator）**：更改对象。例如，`StringBuilder` 类的 `append` 方法，会在原 `StringBuilder` 对象基础上添加内容，改变其状态。

收起

java

```
StringBuilder sb = new StringBuilder("start");
sb.append(" append");
```

这些方法类似于java中的创建Student类后写setname getname setage getage 这些函数来访问类，而不用客户端的人猜name和age是公有还是私有。