4.对象与类：

4.1 面向对象程序设计概述：

面向对象程序设计（OOP），主流程序设计范型，已取代“结构化”过程化程序设计开发技术。`Java`为完全面向对象式程序设计。
面向对象的程序是由对象组成的，每个对象包含对用户公开的特定功能部分和隐藏的实现部分。程序中的许多对象来自标准库，还有一些是自定义。
传统的结构化程序设计通过设计一系列的过程（即算法）来求解问题。确定求解过程之后，开始考虑存储数据的方式。`OOP`将数据放在第一位，然后再考虑操作数据的算法。
对于一些规模较小的问题，将其分解为过程的开发方式比较理想。而面向对象更加适用于解决规模较大的问题。

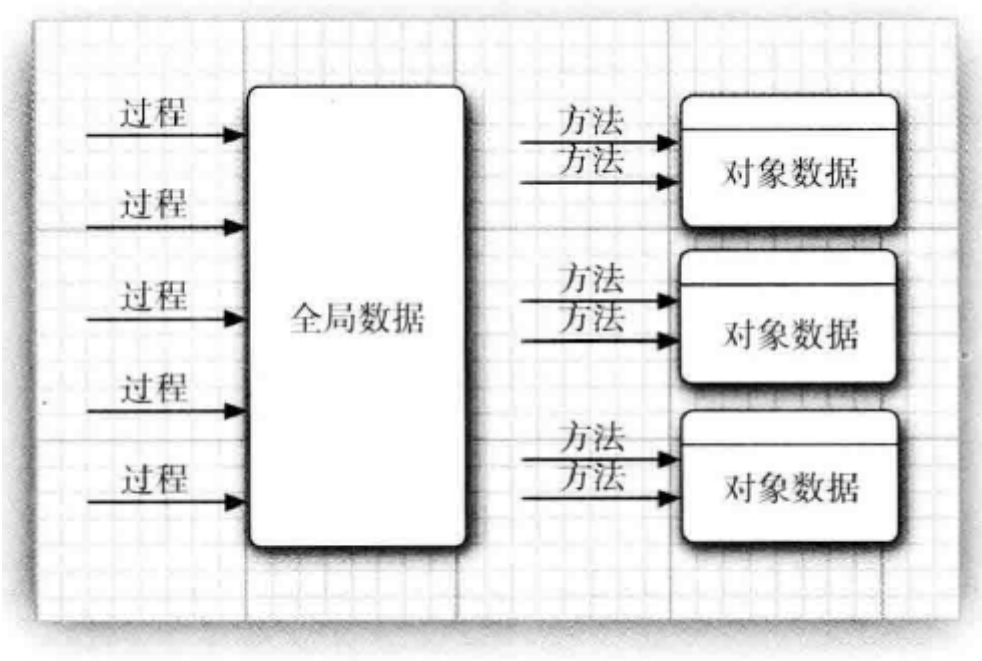

图 4-1　面向过程与面向对象的程序设计对比

4.1.1 类

类（class）是构造对象的模板或蓝图。由类（class）构造对象(object)的过程称为创建类的实例（instance）。

封装（encapsulation），也称数据隐藏。从形式上看，封装是将数据和行为组合在一个包中，对对对象的使用者隐藏了数据的实现方式。对象中的数据称为实例域（instance field），操纵数据的过程称为方法（method）。对于每个特定的类实例（对象）都有一组特定的实例域值。这些值的集合就是这个对象的当前状态（state）。无论何时，只要向对象发送一个消息，它的状态就有可能发生改变。

可以通过扩展一个类来建立另外一个新的类。在`Java`中， 所有的类都源自于一个“ 神通广大的超类”，它就是`Object`。 扩展后的新类具有所扩展的类的全部属性和方法。通过扩展一个类来建立另外一个类的过程称为继承（inheritance)。

4.1.2 对象

对象三个主要特性：

- 对象的行为（behavior)： 可以对对象施加哪些操作， 或可以对对象施加哪些方法？
- 对象的状态（state)： 当施加那些方法时，对象如何响应？
- 对象标识（identity)： 如何辨别具有相同行为与状态的不同对象？

对象的状态并不能完全描述一个对象。每个对象都有一个唯一的身份（ identity)。

4.1.3 识别类：

面向对象程序设计从设计类开始，然后再往每个类中添加方法。

识别类的简单规则是在分析问题的过程中寻找名词，而方法对应着动词。

4.1.4 类之间的关系：

常见关系：

- 依赖（“uses-a”）：一个类的方法操纵另一个类的对象, 尽可能地将相互依赖的类减至最少(让类之间的耦合度最小)。
- 聚合（“has-a”）：聚合关系意味着类`A` 的对象包含类`B` 的对象
- 继承（“is-a”）：用于表示特殊与一般关系. 一般而言， 如果类A 扩展类B, 类A 不但包含从类B 继承的方法，还会拥有一些额外的功能

4.2 使用预定义类

没有类就无法做任何事情，并不是所有的类都具有面向对象特征。
e.g. `Math` 类

4.2.1 对象与对象变量

使用构造器`constructor`构造新实例，构造器是一种特殊的方法，用来构造并初始化对象。
e.g. `Date`类

4.2.2 Java 类库中的LocalDate类

`Data`类的实例有一个状态，即`特定的时间点`。标准`Java` 类库分别包含了两个类：
一个是用来表示时间点的`Date` 类； 另一个是用来表示大家熟悉的日历表示法的`LocalDate`类。

4.2.3 更该器方法与访问器方法

更改器方法（mutator method）：调用方法后，对象状态会发生改变。
访问器方法（accessor method）：只访问对象而不修改对象

4.3 用户自定义类

主力类（workhorse class）, 没有`main`方法，有自己的实例域和实例方法。创建一个完整的程序，应将若干类组合在一起，其中只有一个类有`main`方法。

4.3.1 Employee类

最简单的类定义形式：

```
class ClassName
{
    field1
    field2
    ...
    constructor1
    constructor2
    ...
    method1
    method2
    ...
}
```

源文件名必须与`public`类的名字相匹配。在一个源文件中， 只能有一个公有类，但可以有任意数目的非公有类。

4.3.2 多个源文件的使用

一个源文件包括多个类；或者每一个类存储于单独的源文件中。对于后者，将有两种编译源程序的方法。

- 使用通配符调用`Java`编译器
- 显式地编译高层级的类的源文件

4.3.3 剖析Employee类

关键字`public` 意味着任何类的任何方法都可以调用这些方法. 关键字`private` 确保只有`Employee` 类自身的方法能够访问这些实例域， 而其他类的方法不能够读写这些域。

4.3.4 从构造器开始

构造器总是伴随着`new` 操作符的执行被调用，而不能对一个已经存在的对象调用构造器来达到重新设置实例域的目的。
所有的`Java` 对象都是在堆中构造的，构造器总是伴随着`new` 操作符一起使用。
意在所有的方法中不要命名与实例域同名的变量。

4.3.5 隐式参数与显示参数

方法用于操作对象以及存取它们的实例域。

4.3.6 封装的优点

`getName` 方法、`getSalary` 方法和`getHireDay` 方法, 这些都是典型的访问器方法,由于它们只返回实例域值， 因此又称为域访问器：

```
    public String getName()
    {
       return name;
    }

    public double getSalary()
    {
       return salary;
    }

    public LocalDate getHireDay()
    {
       return hireDay;
```

```
    }
```

4.3.7 基于类的访问权限

4.3.8 私有方法

4.3.9 final 实例域

可以将实例域定义为`final`。构建对象时必须初始化这样的域。也就是说， 必须确保在每一个构造器执行之后， 这个域的值被设置， 并且在后面的操作中， 不能够再对它进行修改。
`final` 修饰符大都应用于基本（primitive) 类型域， 或不可变（immutable) 类的域（如果类中的每个方法都不会改变其对象， 这种类就是不可变的类. 对于可变的类， 使用`final` 修饰符可能会对读者造成混乱.

4.4 静态域与静态方法

4.4.1 静态域

如果将域定义为static, 每个类中只有一个这样的域。而每一个对象对于所有的实例域却都有自己的一份拷贝。

4.4.2 静态常量

静态常量 `System.out`

4.4.3 静态方法

不能向对象实施操作的方法，可以使用对象调用静态方法。
在下面两种情况下使用静态方法：

- 一方法不需要访问对象状态，其所需参数都是通过显式参数提供（例如： Math.pow)
- 一个方法只需要访问类的静态域（例如：Employee.getNextld）

4.4.4 工厂方法

静态方法还有另外一种常见的用途。类似`LocalDate` 和`NumberFormat` 的类使用静态工厂方法(factory method) 来构造对象。

4.4.5 main方法

不需要使用对象调用静态方法,`main` 方法不对任何对象进行操作。

4.5 方法参数

按值调用（ call by value) 表示方法接收的是调用者提供的值。而按引用调用（ call by reference) 表示方法接收的是调用者提供的变量地址。一个方法可以修改传递引用所对应的变量值， 而不能修改传递值调用所对应的变量值。
Java 程序设计语言总是采用按值调用。也就是说，方法得到的是所有参数值的一个拷贝，特别是，方法不能修改传递给它的任何参数变量的内容。

4.6 对象构造

`Java` 提供了多种编写构造器的机制

4.6.1 重载