fileNO 变量中的值异常而导致程序运行错误时，我们只要调查这三个方法就可以了。另外，以后在需要将 fileNO 变量的类型由 int 改为 long 等时，还可以缩小修改造成的影响范围。

OOP 具有将实例变量的访问范围仅限定在类中的功能。加上该限定后的代码如代码清单 4.3 所示。

代码清单4.3　隐藏实例变量

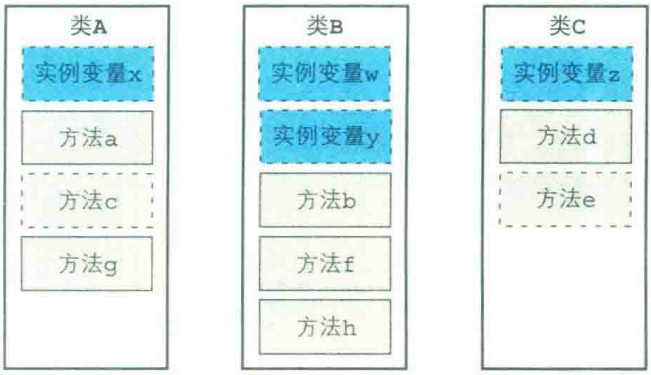
```
class TextFileReader {
  // 存储正在访问的文件编号的变量
  private int fileNO;

  // 打开文件
  //（通过参数接收路径名）
  void open(String pathName) { /* 省略逻辑处理 */ }

  // 关闭文件
  void close() { /* 省略逻辑处理 */ }

  // 从文件读取一个字符
  char read() { /* 省略逻辑处理 */ }
}
```

代码清单 4.2 与代码清单 4.3 只存在细微的差别。后者在实例变量的声明之前添加了 private，这是一种隐藏结构（图 4-3），表示将 fileNO 变量隐藏起来。英文“private”这个形容词的含义为“私人的”“秘密的”。通过该指定，我们可以限定为只有类内部的方法才能访问 fileNO 变量，如此一来该变量就不再是全局变量了。



为了缩小修改的影响范围，我们可以隐藏无法从类外部使用的变量和方法

图 4-3　类的功能之二：隐藏

除了隐藏变量和方法之外，OOP 中还具备显式公开的功能。由于 TextFileReader 类中的三个方法是提供给程序的其他部分使用的，所以我们将其声明为显式公开的方法。修改后的代码如代码清单 4.4 所示。

代码清单4.4　公开类和方法

```
public class TextFileReader {
  // 存储正在访问的文件编号的变量
  private int fileNO;

  // 打开文件
  //（通过参数接收路径名）
  public void open(String pathName) { /* 省略逻辑处理 */ }

  // 关闭文件
  public void close() { /* 省略逻辑处理 */ }

 // 从文件读取一个字符
  public char read() { /* 省略逻辑处理 */ }
}
```

由于在类和方法的声明部分指定了 public，所以从应用程序的任何位置都可以对其进行调用。

> < 类的功能之二：隐藏 >
> 能对其他类隐藏类中定义的变量和方法（子程序）。
> 这样一来，我们在写程序时就可以不使用全局变量了。

4.6　类的功能之三：创建很多个

最后是“创建很多个”的功能。

可能有的读者已经发现了，用 C 语言也可以实现前面介绍的汇总和隐藏功能[1]。然而，使用传统的编程语言则很难实现“创建很多个”的结构，可以说这是 OOP 特有的功能。

下面我们通过示例程序进行讲解。

请大家再看一下代码清单 4.4。它是一个打开文件、读取字符，最后关闭文件的程序。当只有一个目标文件时，这是没有什么问题的，但如果应用程序要比较两个文件并显示其区别，情况会怎样呢？也就是说，需要同时打开多个文件并分别读取内容。我们在代码清单 4.4 中只定义了一个存储正在访问的文件编号的变量。可能有读者会想：“将存储文件编号的变量放到数组中不就行了吗？”请大家放心。即使不进行任何修改，也能同时访问多个文件。

其奥秘就是实例。

我们在第 2 章中介绍过类和实例，还以动物为例，将狗当作类，将斑点狗和柴犬等具体的狗当作实例。

不过，实例并不是直接表示现实世界中存在的事物的结构，而是类定义的实例变量所持有的内存区域。另外，定义了类就可以在运行时创建多个实例，也就是说，能够确保多个内存区域（图 4-4）。

① C 语言中会将源程序分割为多个文件，如果将变量和子程序指定为 static，那么其他文件就无法访问了。

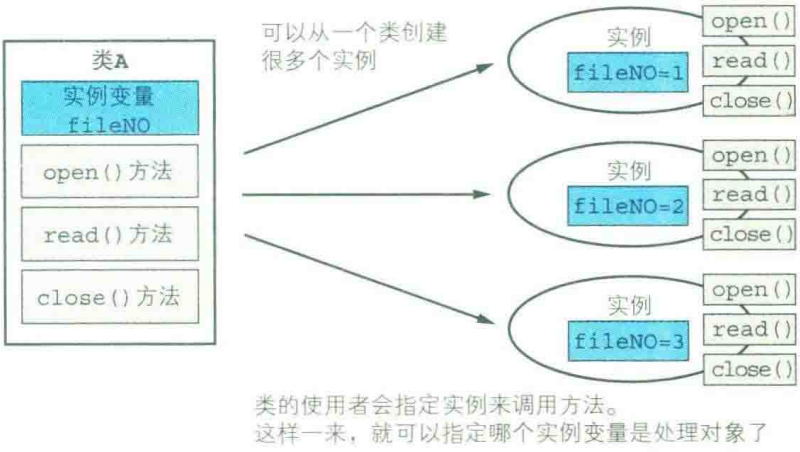


图 4-4　类的功能之三：创建很多个

我们在前面介绍过，类能汇总实例变量和方法。不过，如果同时创建多个实例，那么在调用方法时就不知道到底哪个实例变量才是处理对象了，因此 OOP 的方法调用代码的写法稍微有点特殊。

在传统的子程序调用的情况下，只需简单地指定所调用的子程序的名称。而在 OOP 中，除了调用的方法名之外，还要指定对象实例。根据 Java 语法，应在存储实例的变量名后加上点，然后再写方法名，如下所示。

```
存储实例的变量名 . 方法名 ( 参数 )
```

下面我们就来介绍一下代码清单 4.4 的程序的调用端是什么样子的。请大家看代码清单 4.5。

代码清单4.5　“创建很多个”实例

```
// 从 TextFileReader 类创建两个实例
TextFileReader reader1 = new TextFileReader();
TextFileReader reader2 = new TextFileReader();

reader1.open("C：\\aaa.txt"); // 打开第一个文件
reader2.open("C：\\bbb.txt"); // 打开第二个文件

char ch; // 声明读取字符的变量
```

```
ch = reader1.read(); // 从第一个文件读取一个字符
ch = reader2.read(); // 从第二个文件读取一个字符
ch = reader1.read(); // 从第一个文件读取一个字符

reader1.close(); // 关闭第一个文件
reader2.close(); // 关闭第二个文件
```

这里，首先从 TextFileReader 类创建两个实例，并存储到 reader1 和 reader2 这两个变量中。之后的打开文件、读取字符及关闭文件等处理都是通过指定变量 reader1 和 reader2 来调用方法的。

像这样，通过“指定实例，调用方法”，就可以指定哪个实例变量是处理对象。笔者认为，许多人将 OOP 误解为直接表示现实世界的编程语言，其中一个原因就在于这样的编码方式。也就是说，“指定实例，调用方法”的方式会让人联想起在现实世界中命令属于犬类的斑点狗（实例）抬爪子的场景。不过，我们要将编程结构和为了形象而打的比方区分开。

根据“创建很多个”的结构，类中方法的逻辑就变得简单了。代码清单 4.4 中只编写了一个 fileNO 变量，这意味着定义类的一端完全无须关心多个实例同时运行的情形。传统的编程语言中没有这种结构，所以要想实现同样的功能，就需要使用数组等结构来准备所需数量的变量区域，因此执行处理的子程序的逻辑也会变得很复杂。

一般来说，由于在应用程序中同时处理多个同类信息的情况很普遍，所以这种结构是非常强大的。文件，字符串，GUI 中的按钮和文本框，业务应用程序中的顾客、订单和员工，以及通信控制程序中的电文和会话等，都会应用这样的结构，而 OOP 仅通过定义类就可以实现该结构，非常方便。

< 类的功能之三：创建很多个 >

一旦定义了类，在运行时就可以由此创建很多个实例。

这样一来，即使同时处理文件、字符串和顾客信息等多个同类信息，也可以简单地实现该类内部的逻辑。

以上就是对汇总、隐藏和“创建很多个”这三种功能的介绍。

类结构为编写程序提供了许多便捷功能，但 Java、Ruby 等实际的编程语言都有其各自的功能和详细规范，因此我们可能需要花费一些时间才能充分理解并熟练运用类结构。为了避免在理解时产生混乱，请大家一定要掌握这里介绍的三种功能。

<OOP 的三大要素之一：类 >

类是“汇总”“隐藏”和“创建很多个”的结构。

① “汇总”子程序和变量。

② “隐藏”只在类内部使用的变量和子程序。

③ 从一个类“创建很多个”实例。

4.7 实例变量是限定访问范围的全局变量

下面让我们试着从其他角度来看一下类结构。

如前所述，类结构可以将传统定义的全局变量隐藏为类内部的实例变量。为了更深入地理解类结构与传统结构的不同，我们来比较一下实例变量、全局变量和局部变量。

实例变量的特性如下所示。

< 实例变量的特性 >

① 能够隐藏，让其他类的方法无法访问。

② 实例在被创建之后一直保留在内存中，直到不再需要。

全局变量的问题在于，程序中的任意位置都可以对其进行访问。由于全局变量在程序运行期间一直存在，所以对在超出子程序运行期间仍需管理的信息的保持来说，全局变量是非常方便的。另外，**局部变量**只可以由

特定的子程序访问，只能保持仅在子程序运行期间存在的临时信息。

我们将以上比较结果汇总在表 4-1 中。

表 4-1　三种变量的比较

	局部变量	全局变量	实例变量
多个子程序的访问	×（不可以）	○（可以）	○（可以）
限定可以访问的范围	○（只可以由一个子程序访问）	×（程序的任意位置都可以访问）	○（可以指定仅由同一个类中的方法访问）
存在期间	×（在子程序调用时创建，退出时消除，是临时信息）	○（应用程序运行期间）	○（从实例被创建到不再需要）
变量区域的复制	×（在一个时间点只可以创建一个）	×（每个变量只可以创建一个）	○（运行时可以创建很多个）

也就是说，实例变量融合了局部变量能够将影响范围局部化的优点以及全局变量存在期间长的优点。我们可以将实例变量理解为**存在期间长的局部变量**或者**限定访问范围的全局变量**。

> 实例变量是存在期间长的局部变量或者限定访问范围的全局变量。

另外，实例变量和全局变量一样，在程序中并不是唯一存在的，通过创建实例，能够根据需要创建相应的变量区域。这种灵活且强大的变量结构在传统编程语言中是不存在的（图 4-5）。

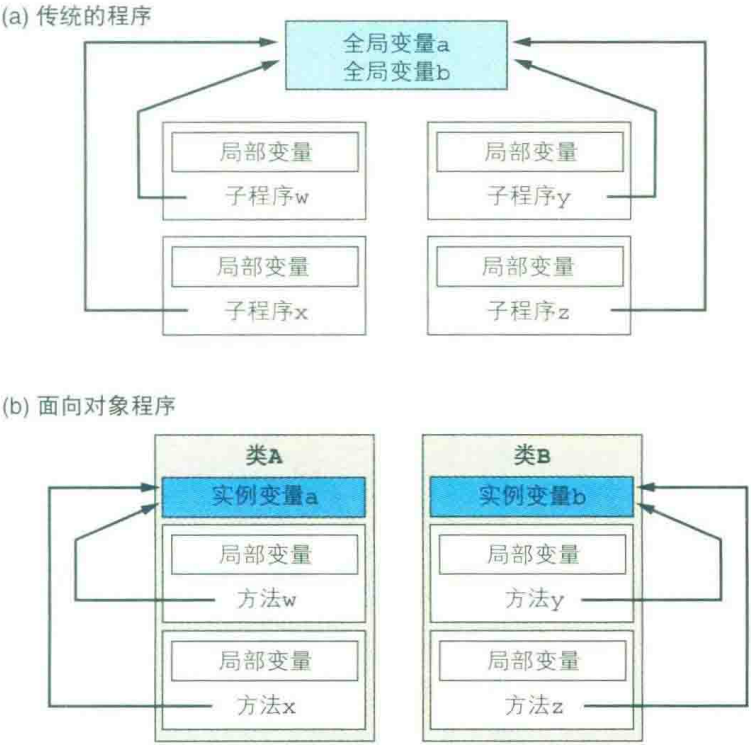


图 4-5　传统的程序和面向对象程序的结构的区别

4.8　三大要素之二：实现调用端公用化的多态

接着我们来看一下三大要素中的第二个要素——**多态**（polymorphism）。顾名思义，多态具有“可变为各种状态”的含义。

在解释多态时，有人会举现实生活中的例子：当发送“哭（叫）”的消息时，婴儿会哇哇地哭，而乌鸦会呱呱地叫。但是，正如我们在第 2 章讨论的那样，面向对象与现实世界是似是而非的。

简单地说，多态可以说是创建**公用主程序**的结构。公用主程序将被调用端的逻辑汇总为一个逻辑，而多态则相反，它统一调用端的逻辑（图 4-6）。

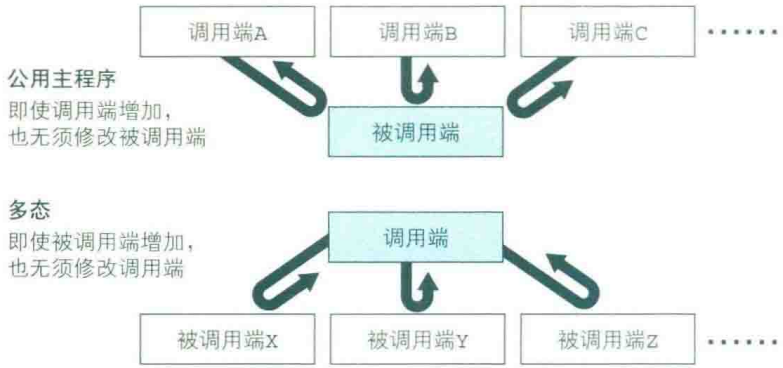


图 4–6　多态的结构

<OOP 的三大要素之二：多态 >

多态是统一调用子程序端的逻辑的结构，即创建公用主程序的结构。

大家可能会觉得“公用主程序”这样的说法有点陈旧，但绝不可小瞧多态。虽说多态只是实现了程序调用端的公用化，但其重要性绝不亚于前面提到的类。在 OOP 出现之前，公用子程序就已经存在了，但公用主程序并没有出现。框架和类库等大型可重用构件群也正是因为多态的存在才成为可能。因此，将多态称为与子程序并列的两项重大发明也不为过。

下面来看一个多态的简单程序示例。我们在前面创建了读取文本文件的类，这次试着创建一个读取通过网络发送的字符串的类，并将该类命名为 NetworkReader（代码清单 4.6）。

代码清单4.6　NetworkReader类

```
public class NetworkReader {
  // 打开网络
  public void open() { /* 省略逻辑处理 */ }
```

```
    // 关闭网络
    public void close() { /* 省略逻辑处理 */ }

    // 从网络读取一个字符
    public char read() { /* 省略逻辑处理 */ }
```

为了使用多态，被调用的方法的参数和返回值的形式必须统一。在代码清单 4.4 中，TextFileReader 的 open 方法的参数指定了文件的路径名，而为了将其与网络处理统一，指定文件的路径名是不恰当的。因此，我们修改一下 TextFileReader 类，在创建实例时指定文件的路径名（代码清单 4.7）。

代码清单4.7　使用多态前的准备

```
public class TextFileReader {
    // 存储正在访问的文件编号的变量
    private int fileNO;

    // 构造函数（创建实例时调用的方法）
    //（通过参数接收路径名）
    public TextFileReader(String pathName) { /* 省略逻辑处理 */ }

    // 打开文件
    public void open() { /* 省略逻辑处理 */ }

    // 关闭文件
    public void close() { /* 省略逻辑处理 */ }

    // 从文件读取一个字符
    public char read() { /* 省略逻辑处理 */ }
}
```

另外，为了使调用端，即公用主程序端无须关注文本文件和网络，我们准备一个新类，并将其命名为 TextReader[1]（代码清单 4.8）。

① TextReader 也可以不是类，而使用仅声明方法规格的接口来实现。

代码清单4.8　TextReader类

```
public class TextReader {
  // 打开
  public void open() { /* 省略逻辑处理 */ }

  // 关闭
  public void close() { /* 省略逻辑处理 */ }

  // 读取一个字符
  public char read() { /* 省略逻辑处理 */ }
}
```

接着，我们在 TextFileReader 和 NetworkReader 中声明它们遵循由 TextReader 确定的方法调用方式。代码清单 4.9 中的 extends TextReader 是继承（后述）的声明，意思是遵循超类 TextReader 中定义的方法调用方式。

代码清单4.9　继承的声明

```
public class TextFileReader extends TextReader {
  // 其他内容与代码清单 4.7 相同
}

public class NetworkReader extends TextReader {
  // 其他内容与代码清单 4.6 相同
}
```

这样就完成了准备工作。通过使用多态结构，无论是从文件还是网络输入的字符，我们都可以轻松地编写出计算字符个数的程序（代码清单 4.10）。

代码清单4.10　使用多态

```
int getCount(TextReader reader) {
  int charCount = 0; // 定义存储字符个数的变量
  while (true) {
```

```
            char = reader.read();  // 使用多态来读取字符
            // 省略满足结束条件时跳出循环的逻辑
            charCount++; // 递增字符个数
        }
        return charCount; // 返回字符个数
    }
```

在代码清单4.10中，getCount方法的参数可以指定TextFileReader或者NetworkReader。另外，即使添加了其他输入字符串的方法，如控制台输入等，也完全不需要对代码清单4.10的程序进行修改（图4-7）。

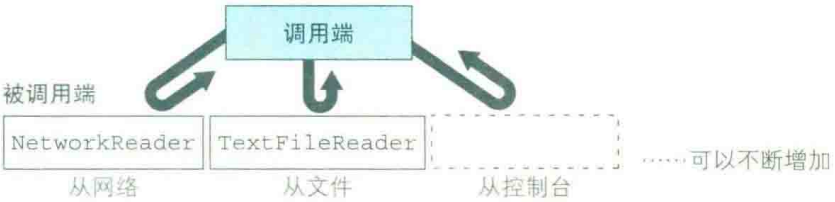


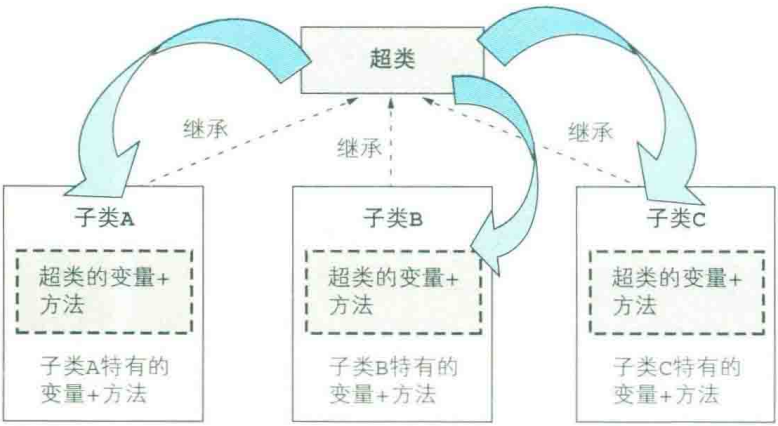
图4-7 利用多态来确保扩展性

4.9 三大要素之三：去除类的重复定义的继承

OOP三大要素中的最后一个要素是**继承**。

我们在第2章介绍过，继承是“整理相似事物的类的共同点和不同点的结构”，相当于数学集合论中的全集和子集。不过，与其说OOP的类结构是对实例进行分类（classify）的结构，倒不如说它是实例的制造装置。

简单地说，继承就是“将类的共同部分汇总到其他类中的结构”。通过利用该结构，我们可以创建一个公用类来汇总变量和方法，其他类则可以完全借用其定义（图4-8）。



子类只声明“继承”就可以定义超类中所有的变量和方法

图 4-8 继承的结构

在 OOP 之前的由子程序构成软件的编程环境中，我们会创建一个公用子程序来汇总重复的命令群。同理，在由类构成软件的 OOP 环境中，我们可以创建一个公用类来汇总变量和方法。也就是说，不仅局限于通过前面介绍的多态来统一调用端，而且还要汇总相似的类中的共同部分。这是一种通过尽可能多地提供功能来让编程变轻松的思想。

在使用继承的情况下，我们将想要共同使用的方法和实例变量定义在公用类中，并声明想要使用的类继承该公用类，这样就可以直接使用公用类中定义的内容。在 OOP 中，该公用类称为**超类**，利用超类的类称为**子类**。

这就好比在现实世界中，孩子从父母那里继承相貌、性情及财产一样，因此有时超类也称为**父类**。

不过，以实际的亲子关系进行比喻很容易引起大家的误解。在 OOP 的继承中，所有性质都会被继承，而在实际的亲子关系中，父母的性质并不会完全遗传给孩子。另外，父母的财产在转让给孩子后就不再属于父母了，而在 OOP 的继承中，超类将其性质传给子类后仍会保留该性质。

因此，抛开这些让人混乱的比喻，我们可以这样理解：继承是将类定义的共同部分汇总到另外一个类中，并去除重复代码的结构。

另外，声明继承也就是声明使用多态[1]。因此，在声明继承的子类中，为了统一方法调用方式，继承的方法的参数和返回值类型必须与超类一致（图 4-9）。

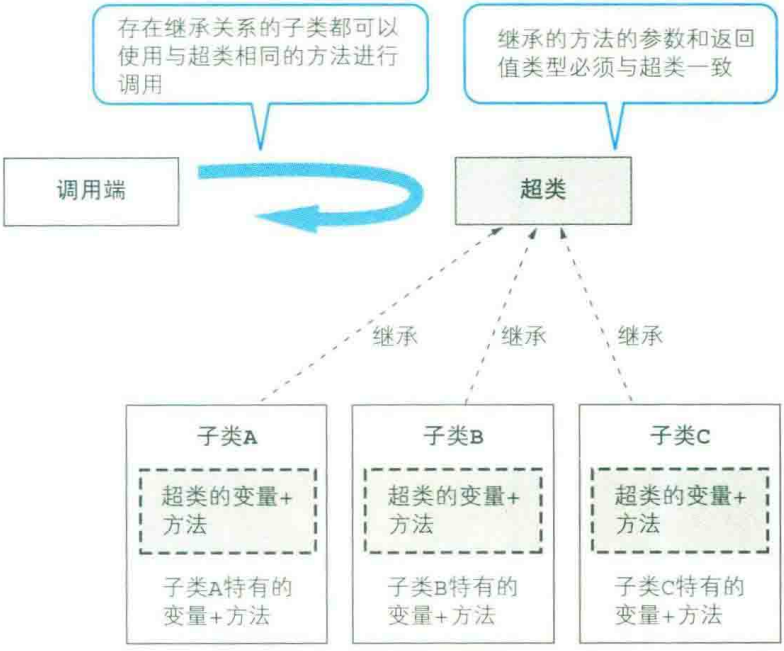


图 4-9　继承和多态

这里省略了继承的示例代码，我们将在第 5 章中详细介绍，感兴趣的读者请参考第 5 章的内容。

下面就让我们来总结一下继承结构。

<OOP 的三大要素之三：继承 >

继承是将类定义的共同部分汇总到另外一个类中，并去除重复代码的结构。

① 这里介绍的继承和多态有时也会分别表示为“实现的继承”和“接口的继承”。

4.10　对三大要素的总结

对 OOP 的三大要素——类、多态和继承的讲解就到此为止，下面我们再来整理一下（表 4-2）。

表 4-2　对 OOP 三大要素的总结

三大要素	类	多　态	继　承
说明	汇总子程序和变量，创建软件构件	实现方法调用端的公用化	实现重复的类定义的公用化
目的	整理	去除冗余	去除冗余
记法	汇总、隐藏和“创建很多个”的结构	创建公用主程序的结构	将类的共同部分汇总到另外一个类中的结构

OOP 之前的编程语言只能通过子程序来汇总共同的逻辑。由于子程序和全局变量是独立存在的，所以很难知道是哪一个子程序修改了全局变量。

OOP 中提供了类结构来解决这个问题。类通过汇总子程序和变量，减少了构件数量，优化了整体效果。再加上多态和继承结构，使子程序无法实现的逻辑的公用化也成为可能。

这三种结构并不是分别出现的，在最初的面向对象语言 Simula 67 中就拥有这三种结构，真是让人惊叹。提起 1967 年，就不得不提到无 GOTO 编程，这真是不平凡的一年。OOP 可以看作结构化语言的发展形式，但考虑到它在那个时代就出现了，因此说是编程语言的突然变异也不为过。

此外，通过组合这些结构还可以实现之前的子程序无法实现的大型重用（关于框架、类库等大规模软件构件群，我们将在第 6 章进行介绍）。