FRC 101 Lecture 2: 面向对象——第一话

Evan Luo

目录

1 前言

1.1 本节课内容

在这节课中，你将会学到：

- 栈（Stack）与堆（Heap）
- 封装（Encapsulation）
- 对象（Object）
- 类（Class）

1.2 如何使用

本文档的前半部分是课程内容的教学。最后一部分是一个 Lab 部分，参考了大学的 Lab 即实验。为了学习效果，在学习完前半部分后，请在一周内完成。

2 开始

2.1 万物皆对象

对象是 `Java` 当中非常重要的概念。其来自于「面向对象」思想，即 OOP（Object-oriented Programming）。面向对象的哲学是：万物皆对象。也就是说我们可以把任何东西看作是一个对象。我们把这样的做法叫做「抽象」。

抽象是哲学、数学中非常重要的思想。抽象的意思是把一件物体的特征提取出来以描述与这件物体相似的一系列同类物体。举个例子，我们从中学开始学习的代数其实就是一种抽象。我们不再专注于学习具体算数的例子，而是学习对于一类数字有效的运算。例如在小学中我们学习 $1+1=2$，而代数则学习 $a+b=c$ where $a,b,c\in\mathbb{R}$

对象在程序中被用于抽象现实中的事物以更好地实现逻辑。比如通过对象，我可以将现实中的学生抽象到程序当中，使得代码可读性更高。

2.2 堆（Heap）与栈（Stack）

在计算机中主要有三种存储数据的位置：RAM（Random Access Memory）、寄存器（Registers；在 CPU 当中）与硬盘（Disk）。其中 RAM 和寄存器只能用于短期的数据存储，如程序运行时的临时变量。因为当程序结束或计算机断电后 RAM 与寄存器当中的数据就会被清空。硬盘则用于持久化的存储，如存储文件等。在断电后，硬盘当中的数据并不会被清空。但在速度上，寄存器和 RAM 远快于硬盘。

堆和栈则是两种存储在 RAM 上的数据结构。需要注意的是，数据结构只是程序存储与读取数据的方式不同，RAM 本身只是一个顺序的列表。由于具体存储方式与我们这节课的内容没有非常大的关系，因此我们只需要专注于了解「什么类型的数据存储与栈或者堆上」。

2.2.1 栈

存储在栈上的数据有一个特点，他们的生命周期（也就是什么时候被创建，什么时候被释放）必须是确定的。并且他们的大小也需要在被分配的时候被确定。也就是说，假如我现在有数据是来自于用户输入，那么就不能用栈来存储。因为在程序编译期间，数据的大小不能被确，直到用户输入时才能确定。而栈在程序已开始就被分配了。因此此类数据不能被存储在栈上。这使得栈丧失了不少的灵活性。一般栈用来存储上节课讲到的基本类型。这类类型的大小都是固定的，生命周期也都贯穿整个程序运行的周期。

但值得注意的是，在丧失了灵活性的同时，栈拥有比堆更好的性能。由于栈上数据的生命周期确定，栈上数据的回收释放并不会触发 GC（Garbage Collector；用于判断一个数据是否需要被回收）。GC 是一个非常耗资源的东西，因此栈通常比堆性能更好。

2.2.2 堆

堆相较于栈，其上的数据不需要确定的生命周期以及大小。因此堆相较于栈更加的灵活。堆一般用于存储对象。由于对象的生命周期不确定，因此需要 GC 来判断数据是否需要清理，这也就降低了堆的性能。（但是现代 Java 在这方面的差距其实已经很小了）

对象中的数据被存储在堆上，那我们怎么去寻找数据具体在哪呢？Java 会在栈上开辟一个空间用来存储「引用」。引用与 C++ 中的指针类似，用于操纵对象。引用将会存储对象在栈上的位置。拿一行代码举例：

```
String s = new String("Hello World!");
```

其中，`String s` 就创建了一个名为 `s` 的引用，其指向真实的对象 `new String("Hello World!")`。

这也就解释了为什么我没有初始化 `s` 的时候程序会报错。因为这个引用没有指向任何存在的对象，也就没有任何意义。

2.2.3 自动装箱与自动拆箱

可以发现在 Java 中，我们可以同时使用 `int` 关键字与 `new Integer(int x)` 创建一个整数类型。

```
int a = 1;
int b = new Integer(1);
Integer c = 1;
Integer d = new Integer(1);
```

以上四种表达都是合法的。但 `int` 是基本类型，`new Integer()` 是对象。我们又是怎么将对象赋值给基本类型（即把对象存储在栈上）的呢？

这涉及到一个语言特性：自动装箱与自动拆箱。

Java 在执行第 2 和 3 行时会自动将对象转为基本类型或将基本类型转为对象。也就是完成了栈到堆或堆到栈的转换。

2.3 类（Class）

万物皆对象，但我们又如何去描述一个对象该有的特征呢？在 Java 中，我们使用类的概念去描述一类对象应该拥有什么特征。举个例子，Java 当中的 `String` 就是一个类。他描述了一个 `String` 对象需要有什么数据和方法。这也就解释了为什么我们在声明对象及其应用的时候使用其类来创建了：`String s = new String("Hello World!");`

类的声明如下：

```
class ClassName {
  // 你可以在类中声明变量。但声明在类中的变量叫做字段（Field）
  int fieldName;

  // 构造器将会在初始化对象时被调用，即使用 new 关键字时
  // 构造器一般用于初始化类中的字段
  // 构造器的声明与方法类似，也同样接受若干参数
  // 但构造器的名字必须与类名同名
  ClassName(int fieldName) {
    // 此处 this 的含义是当前对象的引用。目的是区分对象中的字段与参数（因为此处两者同名）
    this.fieldName = fieldName;
  }

  // 你同样也可以在类中声明一个方法
  void methodName() {
    System.out.println(fieldName);
```

```
  }
}
```

需要注意的是，Java 当中的类需要单独创建一个文件来装，且文件名需要与类名同名。

在创建对象后我们可以通过对象的引用来访问里面的字段以及方法：

```
// 此处叫 clazz 是因为 class 是 Java 关键字，不能在程序中单独作为命名使用。
ClassName clazz = new ClassName(1);
clazz.fieldName = 2;

// Output: 2
clazz.methodName();
```

2.4 封装

封装与复用是写代码过程中非常重要的思想。程序员总是希望以短和清晰的代码去实现一个功能，并且不重复写很多次重复代码（被称作为冗余代码）。实现这个目的的最简单且最常用的方法就是封装。

2.4.1 如何封装

假如说我们现在需要写一个圆的类：

```
class Circle {
  int radius;

  Circle(int radius) {
    this.radius = radius;
  }
}
```

然后在外部我们想要创建一个圆的对象并计算他的面积：

```
Circle circle = new Circle(4);
int area = Math.PI * Math.pow(circle.radius, 2);
```

虽然大部分人知道圆的面积公式，但如果变成像椭圆这样不太常见图形呢？这时候我们就需要封装面积计算的逻辑了：

```
class Circle {
  int radius;

  Circle(int radius) {
    this.radius = radius;
  }

  int area() {
    return Math.PI * Math.pow(radius, 2);
  }
}
```

我们可以通过直接调用 `area()` 来获取圆的面积而不必在每次需要计算面积时都输入一遍公式。一方面减少了程序员对类逻辑理解的负担，一方面又减少了重复的代码。

2.4.2 包

虽然绝大部分情况下你没有必要，但假如某一天你对 Java 自带的 String 类不满意，想要扩充其功能。于是你写了一个自己的 String 类。那么编译器如何去区分你使用的是你自己的 String 类还是 Java 的 String 类呢？这就需要使用到「包」的概念了。

在 C++ 当中使用命名空间来避免这样的冲突，而 Java 当中使用「包」。包类似于文件夹，或者说在文件系统看来就是文件夹。你只需要确保在同一个包下面没有相同名字的类即可。包的声明如下：

```
package net.ironpulse.utils;

class String {
  // Something
}
```

可以看见在第一行我们使用了 package 声明了我们的 String 类位于 utils 这个包下。而 utils 位于 net.ironpulse 下。这样我们在调用时就可以区分了：

```
net.ironpulse.utils.String s = new net.ironpulse.utils.String("Hello World!");
```

但这样很冗长，我们肯定不希望调用的时候需要写下整个包名。此时我们只需要使用 import 关键字，类似于 using namespace std;

```
import net.ironpulse.utils.String;

String s = new String();
```

2.4.3 可见性

通过封装，其他程序员不需要去了解如何去具体实现某个功能，而是只需要知道自己需要调用哪个封装。但假如封装过多也会给程序员造成负担，例如操作系统就有很多「层」封装。

我们常说有底层和高层代码。而大部分人认为底层代码会更难理解。其实不管是底层还是高层代码都是经过了封装的，只不过越到底层封装越多，程序员需要了解的计算机底层原理就越多。这是一种「封装的封装」。前一个封装很显然是给我们自己用的。其他程序员真正需要调用的东西是第二个封装。为了实现只让程序员能够调用后者，我们需要引入可见性的概念。

Java 中有以下几种可见性修饰符：

- public
- protected
- private

可见性修饰符可以被作用于类、方法、字段甚至构造器上。

public 代表任何包下的类都可以访问这个元素。protected 由于涉及到之后的内容将会在下节课涉及到。private 代表只有这个类内部的方法和字段可以访问。以下提供一个例子：

```
class OperatingSystem {
  public launch() {
    enablePower();
    initializeCPU();
    initializeGPU();
  }

  private void enablePower() {
    // Do something
  }

  private void initializeCPU() {
    // Do something
  }

  private void initializeGPU() {
```

```
    // Do something
  }
}
```

可以看见我们对外开放的只有一个 `launch()` 方法。但实际上 `launch()` 的函数中还调用了其他很多的方法。这些方法又是对其他方法的封装。我们通过将这些对外部看来没有用的函数设为 `private` 使得外部知道他们不需要调用这个方法而是只需要调用 `launch()` 即可。

在编辑器当中也是同样的。编辑器当中的提示不会提示 `private` 的字段和类，使得程序员可以快速的找到自己需要的方法或类。

3 作业：Lab 2

写一个类 `Rectangle`，其中需要有两个字段存储长方形的长与宽（单位为厘米）。然后写若干个方法封装计算长方形面积、周长、斜边长（单位需要基于毫米）。最后创建两个 `Rectangle` 对象并将他们两个的面积、周长、斜边长计算结果输出。

提示：单位转换的封装可以使用 `private`。