复习：

1. static：

静态变量：在类加载的时候加载到方法区，在方法区中被赋予默认值。静态变量比对象先出来，所以习惯上通过类名来调用。所有对象存的是这个静态变量的地址，所以所有对象共享同一个静态变量。

静态方法：随着类的加载而加载到方法区，在方法区中只存储不执行，在被调用的时候到栈内存中执行。静态方法先于对象出现，所以习惯上也是通过类名来调用。静态方法中不能使用this/super,也就意味着在静态方法中不能直接使用本类中的非静态。静态方法可以重载，可以被继承，不能重写，可以隐藏。静态方法中不可定义静态变量

静态代码块：用static{}定义，在类加载的时候到栈内存中执行。--- 执行顺序：父类静态 - 子类静态 - 父类非静态 - 父类构造方法 - 子类非静态 - 子类构造方法

注意：能够执行Java程序的内存块只有栈内存。

2. final：

常量：定义好之后值不可变

最终方法：可以被重载可以被继承但是不能被重写/隐藏

最终类：不能被继承

## abstract

如果一个类的所有子类都对这个类中的某个方法做了重写，那么这个时候这个类中的对应的方法可以不定义方法体，需要用abstract修饰方法，从而成为了一个抽象方法。抽象方法所在的类必须是抽象类。--- 抽象类中不一定有抽象方法

抽象类不能创建对象

抽象类一定不是最终类

注意：任何一个类都有构造方法

抽象方法没有方法体，一定要被重写。

抽象方法可以定义被static/final/private修饰吗？---不行

抽象方法一定不能定义在最终类中。

如果一个类中的抽象方法用的是默认权限，对子类有什么要求？--- 要求父子类要同包

练习：定义一个类表示形状，提供获取周长和面积的方法，然后给这个类提供子类：矩形 - 正方形，椭圆 - 圆形

## interface --- 接口

接口中定义都是抽象方法（JDK1.8以前）。类和接口之间用的是implements关键字来产生关联 --- 实现。类在实现接口之后需要重写接口中所有的抽象方法

接口不允许被实例化，也没有构造方法

在Java中，支持的是类和接口之间的多实现 --- 一个类可以实现多个接口

在Java中，支持接口之间的继承，而且接口之间是多继承

从JDK1.8开始，接口中允许定义实体方法 -- 这个是实体方法必须用default修饰，

static方法：只能在本接口中调用，职责上是工具方法，可以通过接口名调用

如果一个接口中只定义了一个抽象方法，那么把这个接口声明为函数式接口，用@FunctionalInteface定义 --- 也是JDK1.8的特性之一

## 内部类

### 方法内部类

定义在方法中的类 --- 方法/局部内部类 --- 为了重复使用某段逻辑，并且使这段逻辑只从属于某一个方法使用

### 成员内部类

定义在类中类

Outer2.Inner2 oi2 = new Outer2().new Inner2();

### 静态内部类

用static修饰的内部类

Outer3.Inner3 oi3 = new Outer3.Inner3();

### 匿名内部类

匿名内部类本质上是实现了对应的接口或者是继承了对应的类

任何一个接口都可以存在匿名内部类形式

一个类只要可以被继承，那么就可以存在匿名内部类形式 --- 最终类不存在匿名内部类形式

扩展：类中可以定义类，类中也可以定义接口，接口中可以定义类，接口中也可以定义接口 --- 如果类中定义了接口或者是接口中定义了接口，那么称之为内部接口 --- 类中定义的接口，以及接口中定义的类和接口默认都是静态的

class A {

static interface A1 {}

}

interface B {

static class B1 {}

static interface B2{}

}

### 包

声明包用的是package --- 区分同名类，进行功能的划分

导入包用的是import --- 导包的作用是用于提示代码从哪儿去找这个类

\* 表示导入当前包下的所有的类但是不包括子包下的类

java --- 原生包

javax --- 扩展包

org --- 第三方厂商提供的一些常用的包

java.lang - 核心/基本包，包含了Java程序运行需要的基本类。在Java程序启动的时候，包下的类就已经自动加载到内存中，所以使用的时候可以不用导包

java.util - 工具包

java.math - 数学运算

java.io - 数据传输

java.net - 网络通信

java.nio - 高并发

java.text - 格式化

总结：java.lang包下的类以及同包类在使用的时候可以不用导包

### 垃圾分代回收机制

针对的是堆内存。

Java中的每种数据类型大小都是确定的，所以所有的内存是由Java自己进行分配，意味着内存的管理和回收也是由JVM自己进行---在Java中一旦产生内存问题导致程序员无法处理。理论上在正常情况下Java中的堆内存是足够使用的 --- 当堆内存使用的负荷量（一般情况下70%）超过一定限度的时候，会启动垃圾回收器（Garbage Collector --- GC）进行堆内存的回收释放

Heap space

Young Generation

eden

survivor

from space

to space

Old Generation

扩展：eden:from:to = 8:1:1

对象刚创建的时候是先放入新生代中的伊甸园区；如果在伊甸园区经过一次回收依然存在，那么将这个对象挪到幸存区，在幸存区中经过多次回收这个对象依然存在则挪到老生代。在回收的时候先回收新生代，如果新生代回收之后的内存足够使用则不扫描老生代；如果不够则扫描老生代。老生代的扫描频率要低于新生代

发生在新生代的回收 --- 初代回收 minor gc

发生在老生代的回收 --- 完全回收 full gc

扩展：对象创建完成之后会先试图放入新生代；如果新生代经过回收之后也放不开，则直接试图将该对象放入老生代。老生代如果也放不开，则会出现错误 --- OutOfMemoryError

API --- Application Programming Interface --- 应用程序接口 --- 接口以及实现类