```
/*
 * JDK1.8之后，接口的语法有改动：
 * 增加了两类成员：
 * （1）静态方法：public static ，static不可以省略
 *  通过“接口名.方法”来进行调用
 *
 * （2）默认方法：public default ，default不可以省略
 *
 * 因为原来API中，往往会设计为这样：
 * （1）集合的接口：Collection
 *     集合的工具类：Collections
 * （2）文件
 *    文件路径的接口：Path
 *    工具类：Paths
 *
 * 觉得没必要为这个接口在增加一个工具类了，干脆把这些方法直接写到接口中，减少.class文件的数量，减少API的类型的数量。
 */
```

```
* 默认方法的冲突问题
* 1、当一个类实现了两个接口，这个两个接口有方法签名相同的默认方法。
* 方法签名：修饰符 返回值类型 方法名（形参列表）
* 如何解决？
* （1）保留其中一个接口的默认实现
* 接口名.super.默认方法(..);
* （2）完全重写
*
* 2、当一个类继承父类，又实现接口，这个类中有与接口的默认方法的方法签名相同方法时
* （1）默认保留的是父类的
* （2）可以选择保留接口的默认实现
* 接口名.super.默认方法(..);
* （3）完全重写
```

```
/*
 * 接口在设计模式中的应用之一：简单工厂模式（了解）
 *
 * 生活：
 * 无工厂：手工
 *    缺点：麻烦，对每个人的要求比较高
 *    例如：衣服，鞋子，需要自己会缝，自己做
 *     优点：个性化
 *
 * 工厂：生产产品
 *    优点：批量生产，使用者比较方便，直接买
 *     缺点：无法定制，如果要定制比较麻烦
 *
 * Java中当创建对象比较麻烦时，或者是需要批量生产对象时，可以交给“工厂类”来生产。
 *
 *
 * 学习难，抽象的知识点：
 * （1）为什么
 * （2）什么意思
 * （3）怎么写
 *
 */
```

```
 * 简单工厂类的作用：生产Java的对象，一般是用来生产某个接口的实现类对象。
 * 好处：把对象的使用者与创建者分离，解耦合（把对象的使用者与具体的实现类解耦合）。
 * 例如：TestSimpleFactory类是对象的使用者
 *       SimpleFactory类是对象的创建者
 *
 * 生活中，生产衣服不是穿衣服的人。
 *
 */
```

```
/*
 * 面向对象的开发原则：
 * （1）面向接口编程
 * （2）对修改关闭，对扩展开发
 *
 * 简单工厂模式：
 * （1）接口
 * （2）一系列的实现类
 * （3）一个工厂类，有一个生产对象的方法
 *
 * 优点：简单
 * 缺点：如果有增加一个接口的实现类，需要修改工厂类的代码
 *
 * 工厂方法设计模式：
 * （1）接口
 * （2）一系列的实现类
 * （3）每一个实现类，会有自己的工厂类
 */
```

```
/*
 * 面向对象的开发原则：
 * （1）面向接口编程
 * （2）对修改关闭，对扩展开发
 *
 * 简单工厂模式：
 * （1）接口
 * （2）一系列的实现类
 * （3）一个工厂类，有一个生产对象的方法
 *
 * 优点：简单
 * 缺点：如果有增加一个产品接口的实现类，需要修改工厂类的代码
 *
 * 工厂方法设计模式：
 * （1）接口
 * （2）一系列的实现类
 * （3）每一个实现类，会有自己的工厂类
 *
 * 优点：如果增加产品的实现类，那么不需要修改工厂类的代码，只需要增加一个对应的工厂类就可以。
 * 缺点：类太多
 */
```

```
/*
 * 静态代理模式：（了解）
 *
 * 代理：
 *   代理可以帮助被代理者完成一些前期的准备工作，后期的善后工作。核心的业务逻辑仍然由被代理者完成。
 *
 * 代码结构：
 * （1）接口：主题
 * （2）被代理者
 * （3）代理者
 * 要求：代理类和被代理类实现同一个主题接口
 *       代理类中要有一个被代理类的属性（target），这样才能把核心业务代码交给被代理者完成
 *
 * 这些需求是和核心业务逻辑无关的代码，而且多变，那么这样的代码可以交给代理完成。
 * 需求：（1）要计算add()的运行时间
 *       （2）记录add()的开始执行，已经结束执行的时间到日志中
 *
 */
```

```
* 1、内部类：
*  当一个类的内部，仍然有一个完整的结构，这个完整的结构仍然需要一个类进行描述，因为有自己的特征（属性，方法），并且这个内部类是为外
*
* 内部类：集合
*
* 身体（Body）里面有各种器官，例如：心脏，也是一个独立的结构，也需要用一个类描述。而且这个心脏是为Body服务，离开Body没有意义。
*
* 2、分类
* 根据位置：
* （一）成员内部类
*  和成员变量一样，在类中，方法外
* 1、静态成员内部类：简称静态内部类
* 2、非静态成员内部类：简称成员内部类
* （二）局部内部类
* 在方法内
* 1、有名字的局部内部类：简称局部内部类（很少）
* 2、没名字的局部内部类：简称匿名内部类
```

```
 * 一、静态内部类
 * 1、如何声明？
 * 【修饰符】class 外部类 【extends 父类】【implements 父接口们】{
 *
 *          【修饰符】 static class 内部类 【extends 父类】【implements 父接口们】{
 *          }
 * }
 *
 * 2、成员
 * 类的5大成员都可以
 * （1）属性：静态和非静态的属性
 * （2）方法：静态的和非静态的方法
```

```
* 一、静态内部类
* 1、如何声明？
* 【修饰符】class 外部类 【extends 父类】【implements 父接口们】{
*
*          【修饰符】 static class 内部类 【extends 父类】【implements 父接口们】{
*          }
* }
*
* 2、成员
* 类的5大成员都可以
* （1）属性：静态和非静态的属性
* （2）方法：静态的和非静态的方法
*       在抽象的静态内部类中，还可以有抽象方法
* （3）代码块：静态的和非静态
* （4）构造器：无参、有参
* （5）内部类：语法上可以，但是太复杂了，不这么写
```

```
* 3、使用
* （1）在静态内部类中不允许使用外部类的非静态的成员
* （2）在外部类中，使用静态内部类和使用其他的类一样
* （3）在外部类的外面使用静态内部类，不需要外部类的对象
* Outer.Inner
* （4）在外部类的外面要调用静态内部类的非静态方法，需要静态内部类的对象
*  例如：Outer.Inner obj = new Outer.Inner();
       obj.test();
* （5）在外部类的外面要调用静态内部类的静态方法，不需要静态内部类的对象
* 例如：Outer.Inner.method();
*/
```

```
* 一、成员内部类（非静态）
* 1、如何声明？
* 【修饰符】class 外部类 【extends 父类】【implements 父接口们】{
*
*          【修饰符】 class 内部类 【extends 父类】【implements 父接口们】{
*          }
* }
*
```

```
* 2、成员
* 类的5大成员都可以，但是不允许有静态成员
* （1）属性：非静态的属性
* （2）方法：非静态的方法
*         在抽象的内部类中，还可以有抽象方法
* （3）代码块：非静态
* （4）构造器：无参、有参
* （5）内部类：语法上可以，但是太复杂了，不这么写
*
```

```
/*
* 三、局部内部类
* 1、如何声明？
* 【修饰符】class 外部类 【extends 父类】【implements 父接口们】{
*
*       【修饰符】返回值类型 方法名(【形参列表】){
*
*           【修饰符】 class 内部类 【extends 父类】【implements 父接口们】{
*           }
*       }
* }
*
* 2、
*/
```