体系结构中立代表的意思是编译后的字节码与特定的计算机体系结构无关

可移植性：Java中int为32位的整数，Java中数据类型有固定的大小，消除了代码移植时的主要问题

每一种平台的解释器是不同的，但是实现的虚拟机是相同的。Java源程序（.java文件）经过编译器编译后变成字节码(.class文件)，字节码由虚拟机解释执行，虚拟机将每一条要执行的字节码送给解释器，解释器将其翻译成特定机器上的机器码，然后在特定的机器上运行。这也就是解释了Java的编译与解释并存的特点。

Java源代码---->编译器---->jvm可执行的Java字节码(即虚拟指令)---->jvm---->jvm中解释器----->机器可执行的二进制机器码---->程序运行。

父类的私有属性和构造方法并不能被继承，所以Contructor也就不能被override,但是可以overload,所以你可以看到一个类中有多个构造函数的情况

String创建之后是不可变的

Java中的值传递和引用传递（非常重要）

Java中总是采用按值传递

**值传递**是指对象被值传递，意味着传递了对象的一个副本，即使副本被改变，也不会影响源对象。（因为值传递的时候，实际上是将实参的值复制一份给形参。）

**引用传递**是指对象被引用传递，意味着传递的并不是实际的对象，而是对象的引用。因此，外部对引用对象的改变会反映到实际的对象上。（因为引用传递的时候，实际上是将实参的地址值复制一份给形参。

接口和抽象类的区别是什么？

接口的方法默认是public，所有方法在接口中不能有实现，抽象类可以有非抽象的方法

接口中的实例变量默认是final类型的，而抽象类中则不一定

一个类可以实现多个接口，但最多只能实现一个抽象类

一个类实现接口的话要实现接口的所有方法，而抽象类不一定

接口不能用new实例化，但可以声明，但是必须引用一个实现该接口的对象 从设计层面来说，抽象是对类的抽象，是一种模板设计，接口是行为的抽象，是一种行为的规范。

**chap3:Java的基本程序设计结构**

final定义常量，常量的命名全部为大写

strictfp关键字标记这个类中的所有方法都使用严格的浮点计算，比如double类型的算式中所有的中间计算都截断为64位

如果运算符得到的值的类型与左侧操作数不同将会发生强制类型转换，如int x=0；x+=3.5；将会得到（int）（x+3.5）

Java中&&和||操作符是按照“短路”方式求值的，如果第一个操作数已经能确定表达式的值，则没有必要计算第二个操作数

而&和|的位操作在得到计算结果之前两个操作数都要计算

当一个字符串和一个非字符串进行拼接时会把后者转为字符串

任何一个Java对象都可以转换成字符串

API：

public final class String//所以String不可变
extends [Object](https://docs.oracle.com/javase/9/docs/api/java/lang/Object.html)
implements [Serializable](https://docs.oracle.com/javase/9/docs/api/java/io/Serializable.html), [Comparable](https://docs.oracle.com/javase/9/docs/api/java/lang/Comparable.html)<[String](https://docs.oracle.com/javase/9/docs/api/java/lang/String.html)>, [CharSequence](https://docs.oracle.com/javase/9/docs/api/java/lang/CharSequence.html)

字符串中==用于确定两个字符串是否放置在同一个位置上

Java中相同的字符串共享，如果复制一个字符串，则原始字符串和复制字符串共享相同的字符

不要用==判断字符串是否相等

码点（code point）表示一个编码表中某个字符对应的代码值

16进制的16位表示为一个代码单元，Unicode字符中基本字符采用一个代码单元编码，辅助字符以及一些奇奇怪怪的字符采用两个代码单元

字符串中只有一个用两个代码单元的奇怪字符时，char ch=sentence.charAt(1)返回的不是空格而是这个奇怪字符的第二个代码单元

所以当字符串中有奇怪字符时，遍历要用codePoints的方法

StringBuilder（单线程编辑）和StringBuffer（效率低但允许多线程编辑）是字符构建器，可通过.toString方法返回字符串

Scanner对象可以用来读取给定文件或字符串，类似之前看的连接串流，例如链接System.in读取控制台输入，有next方法读取输入的单词，nextLine读取输入的一行等

格式化输出中的参数索引值从1开始，如%1$d表示以十进制格式打印第一个参数，具体的格式化标志及转换符参考59页

PrintWriter对象用于写入文件

块：由一对大括号括起来的若干语句，也称为复合语句。块可以嵌套，但是不能在嵌套的两个块中声明同名的变量

while语句先判断再执行，do/while语句先执行再判断，所以最少会执行一次

switch语句从匹配的case标签处开始执行直到遇到break语句，或者执行到switch语句结束，如果没有匹配的case语句而有default子句就执行default子句

goto语句：可以把标签（带：）和break配合使用，跳转到带标签的语句块末尾

continue语句将控制转移到最内层循环的首部

若基本的整数和浮点数无法满足精度要求，可以使用BigInteger（实现任意精度整数运算）和BigDecimal（实现任意精度浮点数运算）类

数组一旦创建就不能再改变大小

for each循环语句更简洁更不易出错

数组长度为0和null不同

Java中实际上没有多维数组，多维数组被解释为“数组的数组”

**chapter4 对象与类**

类之间的常见关系：

1.依赖关系（uses-a）：如果一个类的方法操纵另外一个类的对象，我们就说一个类依赖于另一个类

2.聚合关系（has-a）：类A的对象包含类B的对象

3.继承关系（is-a）：

设计类的时候尽量减少类之间的耦合度，即减少类之间的相互依赖

new Date();会创建一个Date对象，初始化为当前时间

会改变对象状态的方法称为更改器方法，只访问对象而不修改对象的方法称为访问器方法

一个源文件中只能有一个public class

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

构造器总是伴随new操作一起调用，注意不要在构造器中定义与实例域重名的局部变量

注意不要编写返回可变对象的访问器方法，这会破坏封装性，如果需要返回一个可变对象的引用，应该使用clone（）

一个方法可以访问所属类的所有对象的私有数据

private method由于外部的其他类无法访问它，所以程序设计者想把它删去就能删去而不用担心影响到别的类

final类型的实例域必须在构建对象时初始化，且对象构建后就不再被修改

静态域属于类而不是任何独立的对象

静态变量用的少而静态常量用的多，因为通常不想让每个类对象都可以对公有域进行修改

静态方法可以理解为没有this参数的方法

Java中总是采用按值传递，即方法得到的是所有参数值的一个拷贝。注意按值传递在基本数据类型和对象引用中的表现！见书中119页！

重载（overloading）：多个方法有相同的名字、不同的参数

方法的签名包含方法名和参数

仅！！仅当类没有提供任何构造器时系统才会提供一个默认的构造器

注意变量的起名应该易于理解，如aName，aStudent这种

数据域（即head first java中的实例变量）的初始化可以采用在构造器中设置值或直接在数据域中赋值两种方法。还有一种方法称为初始化块，被{}括起来的初始化块会在构造器之前被执行，不常用，作用是可以实现多种构造器的代码复用，因为不管用哪个构造器创建对象都会先执行初始化块。对初始化块标记static就变成了静态初始化块

静态初始化块只在加载类的时候隐式调用一次，初始化块在创建对象时隐式调用

调用构造器的具体处理步骤：

1.所有数据域初始化为默认值

2.按照在类声明中出现的次序依次执行所有域初始化语句和初始化块

3.调用构造器

创建一个子类对象时将调用父类的初始化块和构造函数，子类的构造函数第一行将显式或隐式的添加super（）；这不是创建两个对象，仅创建了一个子对象。父类的初始化块和构造函数被调用是考虑到其可能有私有的属性需要通过自身的构造函数初始化，因为没有创建父类对象，所以父类对象的这些被初始化块和构造函数初始化的属性被存储在堆中和父类动态绑定的区域。

创建一个子类的对象时依次运行：

1.父类的静态初始化块

2.子类的静态初始化块

3.父类的初始化块

4.父类的构造函数

5.子类的初始化块

6.子类的构造函数

可以通过在导入的类前加static关键字导入类的静态方法和静态域：

import static java.lang.Math.\*;

import static java.lang.System.\*;

public class TestStatic {

public static void main(String[] args) {

sqrt(pow(2,2)+pow(2,2));//不用写出Math.sqrt()和Math.pow(),提高可读性，更加简洁

out.println("xixi");//不用写成System.out.println();

}

}

class path中定义了类路径，是所有包含类文件的路径的集合

类设计中应该注意的技巧：

保证数据私有；保证对数据初始化；不要在类中使用过多的基本类型（可以用一个类代替多个基本类型）；不是所有的域都需要独立的域访问器和控制器；将职责过多的类进行分解；类名和方法名应该能够体现它们的职责（例如命名类可以用名词、带形容词修饰的名词、动名词修饰的名词）；优先使用不可变的类（更改对象在多线程时麻烦一点）

**chapter5 继承**

在Java中所有的继承都是公有继承

父类=超类=基类

子类=孩子类=派生类

子类要调用父类的私有域时可以调用父类的公开的方法，用关键字super，如super.getSalary();

super不是一个对象的引用，不能把super赋值给另一个对象变量，它用来调用超类的方法和超类的构造器

this用来引用隐式参数化或调用该类的其他构造器

使用super调用构造器的语句一定是子类构造器的第一条语句

如果子类构造器没有显式地调用超类的构造器，则自动调用超类默认的没有参数的构造器，如果此时发现超类没有默认的没有参数的构造器就会编译错误

调用构造器的语句只能作为另一个构造器的第一条语句，构造参数既可以传递给本类（this）的其他构造器，也可以传递给超类（super）的构造器

多态：一个对象变量实际可以指示多种实际类型的现象

动态绑定：运行时能自动选择调用哪个方法的现象

方法的名字和参数列表称为方法的签名。如果在子类中定义了一个与超类签名相同的方法，那么子类中的这个方法就覆盖了超类中这个相同签名的方法，这时要保证返回类型的兼容性，即允许子类将覆盖方法的返回类型定义为原来返回类型的子类型

private或static或final修饰的方法采用静态绑定的调用方式

覆盖（override）一个方法时，子类方法不能降低超类方法的可见性，如覆盖一个public类型方法时别忘了加public修饰符

覆盖方法时可以在方法头加上@Override来判断是否真的覆盖了原方法

final方法不可被覆盖，final类不可继承，final类的方法会自动变成fianl方法

只能在继承层次内进行类型转换，在将超类转换成子类之前，应该使用instanceof进行检查

包含抽象方法的类必须声明为抽象类

即使不含有抽象方法也能将类设计为抽象的，抽象类不能被实例化。但可以定义一个抽象类的引用对象变量，这个变量只能引用这个抽象类的子类中非抽象类的对象

private：对本类可见

public：对所有类可见

protected：对本包和所有子类可见

default：对本包可见

Java中只有基本类型（数值、字符、布尔类型的值）不是类，所有的数组类型都是类，不管是对象数组还是基本类型的数组都扩展了Object类

equals方法默认要两个对象指向同一块内存区域时才能返回true，自定义的类中应该覆盖这个方法

若子类拥有自己的相等概念，则应该用getClass（a.getClass()==b.getClass()仅在a和b是同一个类时为true）来进行检测

若是由超类来决定相等的概念，就应该用instanceof（(b instanceof A)只要b是A类或A类的子类的实例就会返回true）进行检测，这样可以在不同子类的对象之间进行相等的比较

equals与hashcode的定义必须一致，x.equals（y）返回true的话x.hashCode和y.hashCode必须具有相同的值

Object类的toString方法（public）打印出对象所属的类名和hashcode，覆盖时注意新的方法也要是public的

强烈建议为每个类增加toString方法

ArrayList是泛型类型，具有自动调节数组容量的功能，为ArrayList添加元素使用add方法，set方法只能替换已有元素，如果需替代的位置没有值会报错

Integer、Double这些类作为包装器可用于包装基本类型，这些类是final的，也就是包装在其中的基本类型值是不变的，Java中能实现自动装箱和自动拆箱（由编译器插入必要的方法调用），也就是在诸如int和Interger类之间自动转换

某个对象引用变量=null表示这个变量还未指向某个对象

参数列表中的...表示可以接收任意数量的对象，如public static double max(double...values){}这个方法表示可以接收任意个double参数，double...values在这里等价于一个double[]，这是一个参数数量可变的方法

所有的枚举类型都是enum类的子类，enum可以理解为是与class关键字一个级别

反射用来分析类的能力，其中的Class这个类用来保存某些类的信息（如域、方法和构造器），例如某个ObjectA.getClass方法就会返回一个Class类的实例来保存A这个类的一些信息，199-209中有关反射的内容先跳过

继承的设计技巧：将公有操作和域放置在超类中；不要使用受保护的域；使用继承时考虑是否有（is-a）的关系；除非所有继承的方法都有意义，否则不要使用继承；在覆盖方法时不要改变预期的行为；不要过多使用反射；使用多态而不是类型信息

**chap6 接口、lambda表达式和内部类**

接口中的所有方法都自动的默认为属于public

接口不含实例域，但可以拥有常量

接口不是类，不能实例化，但是可以声明引用接口变量来引用实现了这个接口的类对象（想象成父类变量引用一个子类对象）

接口可以扩展接口，允许有多条从具有较高通用性的接口到具有较高专用性接口的链

元素为类型A的数组要可排序（默认是用归并排序）的话必须满足A这个类型实现了Comparable这个接口（这个接口只有compareTo这个方法），或者有一个compare方法用于数组的排序，总之要有一个compare的标准

比较与前面讨论的equals一样，如果不同子类有不同的比较方法，就应该在compareTo方法之前进行类的检查看是否是同一个类，如果在不同子类之间有通用的比较方法就应该在共同的父类提供compareTo方法并设置为final类型

接口的意义在于提供了多重继承的大多数好处又能避免多重继承的复杂性和低效性

默认方法是接口方法的一个默认实现，标记为default。作用是在接口演化过程中提供源代码兼容性，例如原来某个类a implements某个接口i，后来i中增加了一个方法m，则a因为没有实现方法m会报错，而若将m设为默认方法（有默认实现，默认返回值），则原来的类a不会报错。这意味着接口不一定是100%抽象的。

默认方法冲突（继承的超类和接口中有相同方法）时，对这种二义性的规则是如果是超类和接口冲突则超类优先，如果是接口与接口冲突，则必须覆盖这个方法

希望copy一个新对象，它的初始状态和original相同，但是之后它们会有不同的状态就可以使用clone方法，clone方法是Object父类提供的protect方法（也就是要重新定义clone方法为public的才能允许所有方法克隆对象），但是要实现这个方法必须将这个类implements Cloneable接口，这是一个标记接口，用来确保实现clone方法，否则会抛出CloneNotSupportedException的错误

clone方法默认是浅拷贝，并没有克隆对象中引用的其他对象，所以原对象和克隆对象仍然会共享一些信息，可以重新定义clone方法实现深拷贝

lambda表达式就是一个代码块，形如：（参数）-> {表达式}，最好把它看做是一个函数而不是对象

lambda表达式如果只在某些分支返回值而在另外一些分支不返回值是不合法的

对于只有一个抽象方法的接口，称为函数式接口，当需要这种接口的对象时，就可以提供一个lambda表达式（提高可读性），lambda表达式能传递到函数式接口

在Java SE8 中，接口可以声明非抽象的方法

lambda表达式也可以作为方法或构造器（new方法）的引用，格式为object：：method或class：：method等

lambda表达式可以捕获外围作用域中变量的值，这些能被捕获的变量必须是实际上是最终变量的，也就是在初始化后不会再赋新值

在方法中不能有两个同名的局部变量

可以在类的方法中定义局部类，局部类不能用public或private进行声明，作用域被限定在声明这个局部类的块中，局部类的优势是除了定义它的方法外没有其他方法知道它的权限，而且局部类可以访问事实上为final的局部变量，例如一个可变的int类型变量就不行，而换成一个长度为1的int数组再改变数组里的元素大小就可以正常访问

匿名内部类常用于只创造这个类的一个对象，这个时候就不用命名了。通常的语法为new SuperType（construction parameters）｛

inner class method and data

｝

其中的SuperType可以是接口。匿名类没有名字，也就没有构造方法，构造参数是给SuperType的构造器的。所以如果构造参数的小括号后面跟大括号的就表明正在定义的是匿名内部类

lambda表达式相对而言最简洁

只有内部类可以声明为static类，静态内部类也称嵌套类，在内部类不需要访问外围类对象时应该使用静态内部类，静态内部类可以有静态域和方法

**代理**是Java的基础知识。

代理对象存在的价值主要用于拦截对真实业务对象的访问，代理对象应该具有和目标对象（真实业务对象）相同的方法。要产生一个对象a的代理对象，那么这个对象a必须要有一个接口。

利用代理可以在运行时创建一个实现了一组给定接口的新类。这种功能只有在编译时无法确定需要实现哪个接口时才有必要使用。无论何时调用代理对象的方法，调用处理器的invoke方法都会被调用，并向invoke方法传递Method对象和原始的调用参数。代理类是在程序运行期间创建的，代理类一定是public和final的。

**chap7 异常、断言和日志**

所有异常都由Throwable继承而来，Throwable有Error和Exception两个分支，Exception又有RuntimeException和其他异常两类。Error类和RuntimeException类以及它们都派生类都称为非受查异常（unchecked），所有其他的异常称为受查（checked）异常。

一个方法必须声明所有可能抛出的受查异常，而非受查异常要么是不可控制的（Error），要么是应该避免的（RuntimeException）

子类中覆盖超类的方法时，子类方法中声明的受查异常不能比超类方法更通用，若超类方法不会抛出受查异常，则子类也不能抛出受查异常。

想象以下父类说我不会异常或者只有A异常，子类却会比父类多出异常，这对要用它们的人来说不科学。

习惯上每个派生的异常类都支持一个默认的构造器和一个带有详细描述信息（字符串，可用于Throwable的toString打印）的构造器

包装技术可以让用户抛出子系统中的高级异常，而不会丢失原始异常的细节

带资源的try语句最简形式为

try（Resource res=...）

{work with resource

}

这种形式可以在try块退出时(无论是否抛出异常)自动调用res.close()。带资源的try语句也可以有catch和finally字句，这些会在关闭资源后执行

使用异常机制的技巧：捕获异常的运行时间很长，所以只在异常情况下使用异常机制，异常不能代替简单的测试；不要过分的细化异常；要利用异常的层次结构，寻找或创建更合适的子类异常；不要压制异常；对异常早抛出，晚捕获是可以的

断言机制会在测试期间插入一些检查语句，当代码发布时，这些插入的检测语句会被自动移走。断言只应该用于在测试阶段确定程序内部的错误位置

断言是测试和调试阶段的战术工具，而日志记录是在程序的整个生命周期都可以使用的战略工具

日志管理器在VM启动过程中初始化，这在main执行之前完成。所以main中对日志管理器的设置实际上是在重置

默认情况下日志记录器（Logger）把记录发送到自己的处理器和父处理器中，处理器（Handler）也有日志记录级别。SocketHandle把记录发送到特定的主机和端口上。logger和Handler一起配合工作

日志记录器的命名可以与包名或类名一致

日志默认是将INFO级别及以上的信息记录到控制台，用户可以自定义这些设置

在自定义的类中也应该有toString方法来提供有用的信息，这样可以在调试的时候用print方法帮助调试

还可以为每个类添加main方法来进行单元测试？？？不是很懂，具体的调试技巧还是以后慢慢磨练

**chap8 泛型程序设计（generic programming）**

泛型对于集合类尤其有用，比如ArrayList。泛型可以提高代码的重用性，提供了类型参数（type parameters）。类型参数提高了代码的可读性和安全性

在Java库中用E表示集合的元素类型（element），用K（key）和V（value）表示表的关键字和值的类型，T（有时用临近的U或S）表示任意类型

泛型类可以看作普通类的工厂，用具体的类型代替类型变量就可以实例化泛型类型。如Pair<String>可以看作泛型类public class Pair<T>的实例化

泛型方法中，类型变量放在修饰符后面，返回类型的前面

原始类型的名字就是删去类型参数后的泛型类型名，擦除类型变量并替换为限定类型（不限定的就变为Object）

调用一个泛型方法时，方法名前的<>放入具体的类型

对类型变量进行限制可以用语句<T extends BoundingType>，这里T是绑定类型的子类型，T和绑定类型可以是类也可以是接口，这里的extends关键字用法和之前继承类不同，为了提高效率，应该把没有方法的标签接口放在绑定类型的最后面

Java中泛型转换要记住在虚拟机中没有泛型，只有普通的类和方法；所有的类型参数都会被它们的限定类型替换；桥方法（例如同名父方法a，子方法b，现在想要用b，但类型擦除后只能用a，此时可以生成与a参数相同的方法名c，在c中方法体调用b，这里c就是桥方法）被合成保持多态；为了保证类型安全性，必要时插入强制类型转换

不能用基本类型实例化类型参数，没有Pair<double>,只有Pair<Double>,因为类型擦除为Object，而double这8种基本类型不是Object

运行时类型查询只适用于原始类型，这还是因为类型擦除，例如Pair<String>类型的引用变量用instanceof和getClass都只能得到Pair.class

不能创建参数化类型的数组，还是因为类型擦除会带来bug，如果要收集参数化类型对象，只能使用ArrayList

不能实例化类型变量，也不能构造泛型数组

禁止使用带有类型变量的静态域和方法

不能抛出或捕获泛型类的实例

可以用@SuppressWarnings来消除异常限制

如果Manager是Employee的子类，Pair<Manager>不是Pair<Employee>的子类！！一定要理解，这对于类型安全很重要，例如方法参数为Pair<Employee>，则不能传入一个Pair<Manager>的参数

通配符允许参数类型变化，如Pair<？extends Employee>表示任何类型参数是Employee子类的Pair泛型类型，方法参数为Pair<？extends Employee>时可以传入Pair<Manager>或者Pair<Employee>，它们都是Pair<？extends Employee>的子类！

Pair<？super Manager>限制的是超类型，即类型参数为Manager的父类

**chap9 集合**

队列的常用实现为循环数组和链表（LinkedList）。循环队列更高效但是容量有界

for each循环可以与任何实现了Iterable接口的对象一起工作，Collection实现了Iterable，所以标准类库中的任何集合都能用for each循环

Java迭代器应该认为位于两个元素之间，调用next时，迭代器越过下一个元素，并返回刚刚越过的元素的引用

Iterator接口的remove方法会删除上次访问的对象，必须紧跟在访问一个元素之后执行，如果上次访问之后集合发生了变化，这个方法将抛出错误

数组Array和数组列表ArrayList在增删元素时，后端的所有元素都要跟着挪位置，链表Linkedlist就解决了这个问题，Java中的链表是双向链接的

add只依赖于迭代器的位置，remove依赖于迭代器的状态（即previous/next后再remove的对象不同）

链表不支持快速随机访问，虽然可以用get（
n）方法访问某个特定元素但是效率不高

树集TreeSet是有序集合,可以以任意顺序插入集合，对元素进行遍历时每个值将自动按排序后的顺序呈现，这里排序用的是红黑树结构

优先级队列priority queue中的元素可以按照任意顺序插入，却总是按照排序的顺序进行检索

**chap14 并发**

每个进程拥有自己的变量，而线程共享数据

sleep是Thread的静态方法，会抛出InterruptedException异常

线程用于调用给定目标的run方法

start（）方法用于启动线程，执行其中的run方法

没有可以强制线程终止的方法，不过interrupt方法可以用来请求终止线程，调用interrupt方法在线程的中断状态进行置位，如果线程被阻塞（如调用sleep方法或wait方法等）就无法检测中断状态，这会产生InterrupedException异常

线程有6种状态：new、runnable、blocked、waiting、timedwaiting、terminated

调用start方法使thread进入runnable状态，可运行状态的线程可能在运行也可能没在运行

一个线程a创建的线程b（a是b的父线程，b是a的子线程）拥有同样的优先级，而且只有当a是守护进程时b才是守护进程

默认的情况下线程继承它的父线程的优先级，优先级在1（最低MIN\_PRIORITY）-10（最高MAX\_PRIORITY）之间，NORM\_PRIORITY为5。

线程的优先级高度依赖于系统，例如在win中java的10个优先级被映射到7个优先级，在linux中10个优先级变成同样的优先级，所以不要将程序构建为功能的正确性依赖于优先级

yield方法使当前线程处于让步状态，在其他可运行线程的优先级至少与这个线程同样高时将先运行其他可运行线程

守护线程Daemon的唯一用途就是为其他线程提供服务，当只剩下Daemon时虚拟机就退出了

线程退出时会判断自己是否是当前整个JVM中最后一个非Daemon线程，如果是就卸载JVM(过程简化不说了)

线程遭遇未捕获异常时，异常被传递到一个用于未捕获异常的处理器

由于线程是共享数据的，所以会产生竞争条件，所以同步很重要

公平锁比常规锁要慢很多

ReentrantLock是可重入锁，可重入表示可以重复获得已经持有的锁。锁是由线程获得的而不是具体的线程内的同步方法获得的。所以对于可重入锁，某个线程获得该锁后这个线程的其他需要这个被锁住对象的方法也都能获得锁并执行。

条件对象也称条件变量，锁可以拥有一个或多个相关的条件对象

在一个线程拥有某个条件的锁时，它仅仅可以在该条件调用await、signalAll、signal方法。例如在线程a中，锁的条件的await方法是在不满足某个条件时退出锁并且阻塞，将线程放到条件的等待集中，直到某个别的线程用signal或者signalAll方法通知a这个条件可能改变了，然后a退出阻塞状态重新获得竞争锁并测试条件的资格。

signalAll是解除该条件等待集中所有线程的阻塞状态，signal是随机解除条件等待集中一个线程的阻塞状态

要小心死锁。某个线程await了以后阻塞了，如果别的线程没有用signal或者signalAll方法通知它，它就不再运行了，如果所有别的线程也都await了那就没有任何线程能解除其他线程的阻塞了。

synchronized关键字用了内部对象锁，内部锁有一个内部条件。wait和notifyAll和notify方法和锁中的条件的await、signalAll、signal方法理解相同。wait和notifyAll和notify方法只能在同步方法中调用，并且如果当前线程不是对象锁的持有者会抛出IllegalMonitorStateException异常

锁是锁在对象上的，synchronized的方法要获得对象锁才能调用，没有锁的方法不用获得锁。例如如果一个对象A有synchronized的方法a和synchronized的方法b和一个没有synchronized的方法c。然后一个线程正在执行a，那另外一个线程能执行c但是不能执行b。一个对象下所有synchronized的方法所需要的是同一把锁，就是这个对象锁。

静态方法声明成synchronized也是合法的，这获得了类对象的内部锁，使其他线程无法调用这个类的同步静态方法

除了方法外我们可以对部分代码进行同步化用synchronized关键字修饰也能获得对象锁

volatile关键字为实例域对同步访问提供了免锁机制

要仔细设计避免出现死锁

可以用ThreadLocal辅助类来帮助创建线程局部变量

队列数据结构比如阻塞队列也能用来作为一种同步机制。阻塞队列是线程安全的集合

并发就是为了线程间共享数据的安全性？高效利用?不是的，真正目的是为了充分发挥计算机的计算性能。

有一些线程安全的集合可以完成诸如原子更新的操作

ArrayList和HashMap不是线程安全的，不过任何集合类都可通过使用同步包装器变成线程安全的

如List<E> synchArrayList=Collections.synchronizedList(new ArrayList<E>())就用锁对集合进行保护

Runnable和Callable都是异步运行的任务，区别在于Runnable没有返回值而Callable有返回值。Callable接口是一个参数化的类型，类型参数是返回值的类型，只有一个方法call。

如果要创建许多生命期很短的线程，可以用执行器Executor来构建线程池。线程池中的空闲线程调用完run方法后不会死亡而是等待下一个服务请求。

线程池的操作流程是创建线程池、提交任务、关闭线程池。

Swing不是线程安全的

除了时间分配进程不要在其他线程中接触Swing组件

Swing编程中的单一线程原则