

# Chapter 1 java简介

---

## 1.1 java历史以及特征

- 由Sun公司推出的面向对象的语言，具有安全性，简单性，易用性， **平台无关性**
- 最大的优势是跨平台运行，源代码 -> .class文件

## 1.2 API, JDK, IDE

- API: application program interface； 程序接口
- JDK: Java Development Kit； 包含软件库，基于命令行的编译器等
- IDE: Integrated Development Environment

## 1.3 HelloWorld

参见Coding.ch1.HelloWorld

## 1.4 jvm

java虚拟机，全名java virtual machine，字节码文件（.class文件）可以在任何装有jvm的pc上运行，jvm是java跨平台运行的根本原因，它有一套虚拟的CPU指令集和汇编指令，.class文件包含了JVM的CPU指令集。

## 1.5 安装JDK

这里不加赘述，懂得自然懂。

# Chapter 2 基本程序设计

## 2.1 编写简单的程序

编写程序，给定半径，计算圆的面积

```
package www.learnjava.garfield.ch2; // java的类都在包中
// 默认导入 java.lang.System类
public class Area {
    public static void main(String[] args) {
        double radius = 10;
        double area;
        double pi = 3.1415;
        area = radius * radius * pi;
        System.out.println(area);
    }
}
```

## 2.2 简单IO

- 标准输入/输出流
  - System.out: 标准输出流类OutputStream的对象
  - System.in: 标准输入流类InputStream的对象
- Scanner类 (java.util.Scanner)

```
Scanner scanner = new Scanner(System.in);
// 从控制台读入一个double类型数
double d = scanner.nextDouble();
// 从控制台读取一个字符串
String s = scanner.next();
// Scanner还有很多方法，如nextInt, nextByte等
```

## 2.3 标识符，变量，常量

### 2.3.1 标识符

标识符被用来命名常量，变量，方法，类等实体，命名规则有以下几点：

- 由字母、数字、下划线、美元符构成
- 开头不能是数字
- 不能是保留字

### 2.3.2 变量

变量用于保存数据输入，输出，中间值等，变量声明语法: `datatype variableName`, 例如 `int x`。

### 2.3.3 常量

常量一旦初始化后就不能再改变，常量声明语法: `final datatype CONSTANT_NAME = value`，注意常量的声明和初始化必须同时完成。

## 2.4 赋值语句和赋值表达式

- 赋值语句

形如 `variable = expression` 的语句，其中 `expression` 是包含字面量、变量、方法调用和操作符的表达式。赋值语句的结果是将表达式的值赋值给左边的变量。例如 `x=1`

- 赋值表达式

赋值表达式形式和赋值语句相同，赋值表达式的结果等于表达式的值，赋值表达式是右结合的。例如 `i=j=k=1`。

## 2.5 JAVA数据类型

### 2.5.1 基本数据类型

- 整数类型: `byte, short, int, long`
- 字符类型: `char`
- 浮点类型: `float, double`
- 逻辑类型: `boolean`

### 2.5.2 引用类型

- 类
- 接口
- 数组

### 2.5.3 数值字面值 (literal)

定义：字面值是直接出现在程序中的常量值。例如 `long l = 10000L` 中的 `10000L` 就是字面值。

#### 整数字面值

- 以0开头表示八进制
- 以0x或者0X开头表示十六进制
- 以1-9开头表示十进制
- 以l或L结尾表示long类型
- 无后缀表示int类型

#### 浮点数子字面值

- 以d或D结尾或者无后缀表示double类型
- 以f或者F结尾表示float类型

## 2.5.4 数值类型转换

如果二元操作符的两个操作数的数据类型不同，那么根据下面的规则对操作数进行转换：

- 数据转换总是向较大范围的数据类型转换，避免精度损失。

```
byte i = 10;
long k = i*3 + 4; // i转成int参与表达式计算，计算结果转long
double d = i*3.1 + k/2; // i转成double, k/2转double
```

- 将值赋值给较大取值范围的变量时，自动进行类型转换。

byte < char < short < int < long < float < double

- 将值赋值给较小取值范围的变量时，必须进行强制类型转换。

```
float f = (float)10.1; // 10.1默认double类型, double>float
int i = (int)f;
```

### 注意

- 整数操作时，除数不能为0，否则会产生ArithmaticException异常
- 浮点数操作上溢至Infinity，下溢至0  
浮点数除0等于Infinity  
0.0除0.0等于NaN

## 2.5.5 字符数据类型

- char表示16位的单个Unicode字符（Java中字符都是Unicode编码）
- char类型的字面值
  - 以两个单引号界定的单个Unicode字符。如'A'
  - 可以用\uxxxx形式表示，xxxx为16进制。如\u7537
  - 可以用转移字符表示：\n \t \b等
- String表示一个字符序列，是基于String类实现的，不是java内置的数据类型，是引用类型

## 2.6 编程风格和常见错误类型

### 2.6.1 编程风格

- 注释：
  - 类和方法前使用javadoc文档注释
  - 方法步骤前使用行注释
- 命名：
  - 类名首字母大写，驼峰命名
  - 变量和方法名小写，多个单词，第一个单词首字母小写，其他单词首字母大写
  - 常量使用大写，单词间以下划线分割
- 缩进为4个空格

## 2.6.2 java常见错误类型

- 语法错误 (syntax error) : 编译期间产生的错误, 编译报错
- 运行时错误 (runtime error) : 导致程序非正常终止的错误, 编译不会报错
- 逻辑错误 (logic error) : 程序不能按预期的方式执行, 编译不会报错

# Chapter 4 数学函数、字符和字符串

## 4.1 常用数学函数

Java中有一个Math类，是一个final类，再java.lang.Math包中，所有数学函数都是静态方法。

- Math类中定义了常用的数学常量
  - PI: 圆周率
  - E: 自然对数

- Math类中定义了常用的数学方法
  - 三角函数: sin, cos, tan...
  - 指数: exp, log, log10, pow, sqrt...
  - 取整: ceil, floor, round
  - 其他: min, max, abs, random( [0.0,1.0] )

通过Math.random可以实现随机生成字符，代码参见  
Coding/.../ch4/RandomCharacter。

## 4.2 字符串数据类型和操作

### 4.2.1 Unicode 和 ASCII

Unicode包括ASCII码，从'\u0000'到'\u007f'对应128个ASCII字符，java中的ASCII字符也可以用Unicode表示，例如 `char a = '\u0041'` 等价于 `char a = 'A'`。

### 4.2.2 字符型数据和数值类型数据之间的转换

char类型数据可以转换成任意一种数值类型，反之亦然。将整数转换成char类型数据时，只用到该数据的低16位，其余被忽略。如 `char ch = (char)0xAB0041`，其中高位AB被忽略掉，只将0041进行转化，ch值为A。将浮点数转成char，先将浮点数转为int型，再转为char。如果转化不产生精度损失，可以采用隐式转换，否则必须强制类型转换。

### 4.2.3 Character类

Character类是char的包装类，它的作用在于：将char类型的数据封装成对象；包含处理字符的方法和常量。该类有对字符进行类型判断和对字符进行大小写转换的常用方法。

## 4.3 字符串类型

### 4.3.1 String类

String类是一个final类，不能被继承，表示一个固定长度的字符序列，实例化后其内容不能更改。不能更改不是真正意义上的不能更改，而是一旦更改了，原来的字符串引用次数为0，那就被垃圾回收了。

String类的实例化：

```
// 从字面值创建字符串
String message = new String("Welcome to Java");
// 简写形式
String message2 = "welcome to Java";
```

### 4.3.2 规范字符串和常量池

```
String m1 = "welcome";
String m2 = "welcome";
String m3 = "We1" + "come";
String m4 = "We1" + new String("come");
m2 == m1;           // true
m1 == m3;           // true
m1 == m4;           // false
```

通过上例可以感觉到java的String存储机制有悖我们的直觉，是因为java中存在常量池。为了提高效率和节省内存，Java中的字符串字面值维护在字符串的常量池中。

实际上上面的m1和m2和m3指向的都是常量池中的“Welcome”，因此它们都指向的是相同的地址，而m4中new String("come")不是一个字面值，因此m4指向的时堆中的一个地址。

### 4.3.3 字符串常用方法

- String.equals(String): 两个字符串比较
- String.equalsIgnoreCase(String): 两个字符串比较，忽略大小写
- String.regionMatch(int, String, int, int): 比较两个字符串的一部分
- String.startsWith(String): 判断是否以某个字符串开头
- String.endsWith(String): 判断是否以某个字符串结尾
- String.length(): 获取字符串长度
- String.charAt(int): 获取字符串在某个位置的字符
- String.substring(int, [int]): 字符串截取，返回新字符串
- String.toLowerCase(): 转成小写，返回新字符串
- String.trim(): 删除首尾空格，返回新字符串
- String.replace(String, String): 字符串替换，返回新字符串
- String.indexOf(char): 查找某个字符的首次出现位置，返回-1表示未找到

### 4.3.4 StringBuilder和StringBuffer

String类一旦初始化完成，字符串就不可修改，但是StringBuilder和StringBuffer初始化后还可以修改字符串，StringBuffer修改缓冲区的方法是同步的，更适合多线程环境。StringBuilder线程不安全，与StringBuffer工作机制类似。StringBuilder和StringBuffer还拓展了字符串的方法，例如append, insert, reverse等。

## 4.4 格式化控制台输出

JDK1.5之后支持 `System.out.printf()` 格式化输出字符串。举例如下：

```
public static void main(String[] args) {  
    System.out.printf("boolean:%6b\n", false);  
    System.out.printf("boolean:%6b\n", true);  
    System.out.printf("character:%4c\n", 'a');  
    System.out.printf("integer:%6d,%6d\n", 100,200);  
    System.out.printf("double:%7.2f\n", 12.23);  
    System.out.printf("String:%7s\n", "JAVA");  
}
```

# Chapter 5 Loop

## 5.1 while循环

语法:

```
while(condition){  
    // do something...  
}
```

## 5.2 Do-While循环

语法:

```
do{  
    // do something  
} while(condition)
```

## 5.3 For循环

语法:

```
for(initial action; condition; iteration){  
    // do something  
}
```

## 5.4 增强For循环

语法:

```
for(elementType val : arrayRef){  
    // do something  
}
```

## 5.5 Break和Continue

- break: 结束当前循环
- continue: 跳过当前循环



# Chapter 6 方法

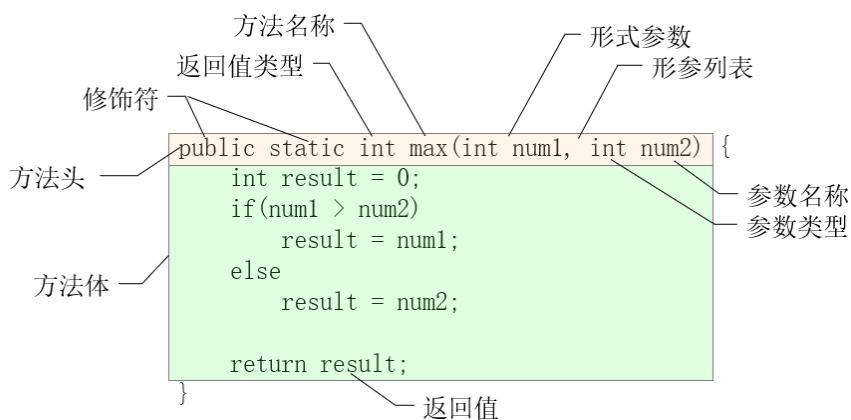
## 6.1 方法的定义

定义：方法（method）是为执行一个复杂操作组合在一起的语句集合。一个类中可以声明多个方法。

语法：采用BNF（巴科斯范式）描述

方法定义示例：

### 方法定义示例



注意：

- 一个类中不能包含方法签名相同的多个方法，方法签名指方法名称+形参列表（不含返回类型）
- 方法中形参可以用final修饰，表示方法内部不允许修改该参数
- 形参不允许有默认值，最后一个形参可以为变长参数
- 方法里不允许定义static局部变量
- 方法可以有一个返回值，也可以没有返回值，但构造函数没有返回值，也不能加void

## 6.2 方法调用

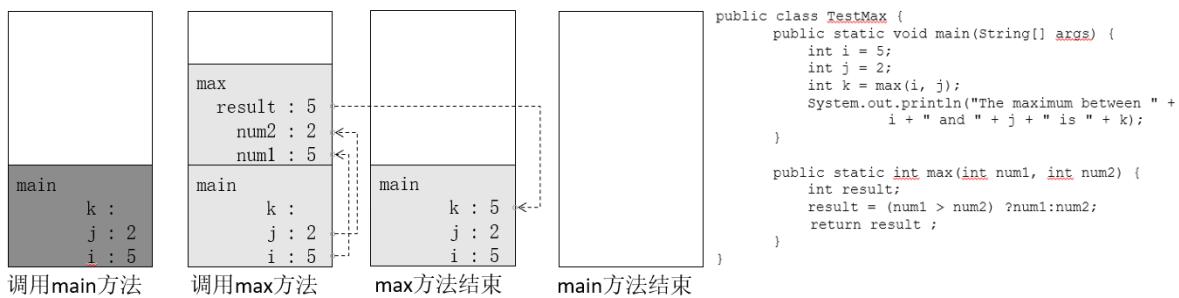
### 6.2.1 静态方法和实例方法

方法分为静态方法和实例方法。

- 实例方法：依靠实例而存在的方法
  - 调用当前类中的实例方法：通过 方法名 或者 通过this.方法名
  - 调用其它类中的实例方法：必须实例化其他类，再通过这个 实例.方法名 调用
- 静态方法：依靠类而存在的方法，也可以通过实例访问
  - 调用当前类中的静态方法：通过 方法名 或者 通过类名.方法名
  - 调用其他类中的静态方法：通过 类名.方法名 或者 通过 对象名.方法名

## 6.2.2 调用堆栈

每当调用一个方法，系统将方法参数、局部变量存储在一个内存区域中，该内存区域称之为调用堆栈，当方法结束返回给调用者时，系统会自动释放想用的调用栈。



## 6.3 方法参数传递

- 当形参为call by value类型，在方法内修改形参不会影响到实参
- 当形参为call by reference类型，在方法内修改形参会影响到实参
- 实参类型、顺序必须和形参类型、顺序一致
- 类型兼容：子类实参可以传递给父类形参

## 6.4 方法的重载（Overload）

方法重载指方法名称相同，但形参列表不同的方法，仅返回类型不同的方法不是合法的重载，一个类中可以包含多个重载的方法。

歧义调用：调用时匹配成功的方法可能多于一个，则会产生编译二义性

```
public class AmbiguousInvocation {  
    public static void main(String[] args) {  
        System.out.println(AmbiguousInvocation.max(1, 2));  
        // 编译器报错：对max的引用不明确，无法确定用哪个函数，因为参数都能相容  
    }  
  
    public static double max(int num1, double num2){  
        return (num1 > num2)?num1:num2;  
    }  
  
    public static double max(double num1, int num2){  
        return (num1 > num2)?num1:num2;  
    }  
}
```

# Chapter 7 数组

## 7.1 基础

定义：数组是相同类型的集合。

初始化：通过 new 关键字初始化，凡使用new后，内存单元都初始化为0或者null或者false或者'\u0000'。

注意：声明数组引用变量并不分配数组内存空间，必须通过new实例化数组来分类数组内存空间。

## 7.2 数组的复制

由于数组是引用类型，通过赋值语句不能实现对数组的深拷贝，因此复制数组的方法有以下几个：

- 使用循环来赋值每一个元素
- 使用 `System.arraycopy`，前提：两个数组都与先实例化了
- 使用数组的clone方法复制，被复制的数组变量可以没有实例化。

## 7.5 可变长参数

在方法中，最后一个形参可以设置成可变长参数，表示形参个数不定，java可以将可变长参数当作数组看待。

```
public class VariableLengthParameter {  
    public static void main(String[] args) {  
        System.out.println(VariableLengthParameter.sum(1, 2, 3, 4, 5, 6, 67, 78));  
    }  
    public static int sum(int n1, int n2, int... n3) {  
        int s = 0;  
        s += n1;  
        s += n2;  
        for (int i : n3) {  
            s += i;  
        }  
        return s;  
    }  
}
```

上例中我们可以看到sum方法看似形参只有3个，但实际上可以传大于3个的任意个数的参数，这就是可变长参数，实际上就是数组的变形，我们可以通过for循环遍历可变长参，也可以通过 length 方法获取可变长参数的长度。

## 7.6 数组的查找和排序

### 7.6.1 查找

- 线性搜索：顺序遍历，最坏情况下比较N次，平均比较N/2，时间复杂度为O(N)
- 二分查找：在一个已排序好的数组中进行查找，时间复杂度为O(logN)

```
public static int binarySearch(int[] arr, int target) {  
    int len = arr.length;  
    int low = 0;  
    int high = len - 1;  
    while (low < high) {  
        int mid = (low + high) / 2;  
        if (arr[mid] == target) {  
            return mid;  
        } else if (arr[mid] > target) {  
            high = mid - 1;  
        } else {  
            low = mid + 1;  
        }  
    }  
    return -1;  
}
```

### 7.6.2 排序

- 选择排序
- 冒泡排序
- .....

## 7.7 Arrays类

java.util.Arrays类中实现了常见的排序和搜索方法，我们可以直接调用API即可

```
int[] a = new int[]{3,4,2,8,5,6,956,345,5645};  
Arrays.sort(a);  
int index = Arrays.binarySearch(a, 956);  
System.out.println(Arrays.toString(a)); // [2, 3, 4, 5, 6, 8, 345, 956,  
5645]  
System.out.println(index); // 7
```

## 7.8 命令行参数

我们可以从命令行向java程序传递参数。参数以空格分隔，如果参数本身包含空格，可以用双引号括起来。

格式：`java 类名 参数1 参数2`

例子：`java TestMain "First Arg" aaa 123`

命令行参数对应的就是main方法中的`String[] args`，可以通过访问args来实现对每个参数的访问。

## 7.9 多维数组

声明二维数组引用变量： `dataType[][] refVar;`

创建数组并赋值给引用变量：当指定了行、列大小，是矩阵数组（每行的列数一样）。非矩阵数组则需逐维初始化。

```
refVar = new dataType[rowsize][colsize]; (这时元素初始值为0或null)
```

# Chapter 9 对象和类

## 9.1 对象和类的定义

对象：现实世界中可识别的**实体**，具有一定的状态和行为

类：是对同种对象（对象具有相同的属性或者方法）的**抽象**

举例：我是一个人，你也是一个人，那么人就是一个类，而我和你就是这个类的实例，是对象。

## 9.2 定义类、实例化类

当创建对象数组时，数组元素的缺省初值为null。

```
circle[] circleArray = new Circle[10]; //这时没有构造Circle对象，只是构造数组
for(int i = 0; i < circleArray.length; i++) {
    circleArray[i] = new Circle(); //这时才构造Circle对象，可使用有参构造函数
}
```

## 9.3 理解构造函数

构造函数特征：

- 无返回类型
- 名字同类名
- 用于初始化对象
- 只在new时被自动执行

构造函数限制：

- 必须是实例方法（无static），可以为共有、保护、私有和包级权限
- 如果没有定义任何构造函数，编译器会自动提供一个不带参数的默认构造函数
- 如果已自定义构造函数，则不会提供默认构造函数

## 9.4 理解对象访问

## 9.5 对象的变量、常量和方法

### 9.5.1 实例变量和静态变量

- 实例变量：未用static修饰的成员变量，属于类的具体实例，只能通过对象访问
- 静态变量：用static修饰的变量，被类的所有实例所共享，可以通过实例访问，也可以直接通过类访问

## 9.5.2 实例常量和静态常量

- 实例常量：没有用static修饰的final变量
- 静态常量：用static修饰的final变量
- 所有常量可以按需指定访问权限，但由于他们不能被修改，所以通常定义为public

## 9.5.3 类和实例的方法

### 静态方法

- 用static修饰的方法
- 构造函数不能用static修饰
- 静态方法没有this引用
- 静态方法可以通过对象和类名调用
- 静态方法内部只能访问类的静态成员 (因为实例成员必须有实例才存在，当通过类名调用静态方法时，可能该类还没有一个实例)
- 静态方法没有多态性

### final修饰

- final修饰实例方法时，表示该方法不能被子类覆盖（Override）。非final实例方法可以被子类覆盖
- final修饰静态方法时，表示该方法不能被隐藏（hiding）。非final静态方法可以被子类隐藏
- 构造函数不能为final

### 方法重载（overload）

同一个类中、或者父类子类中的多个方法具有相同的名字，但这些方法具有不同的参数列表(不含返回类型，即无法以返回类型作为方法重载的区分标准)

### 方法覆盖（override）和方法隐藏（hiding）

发生在父类和子类之间，前提是继承。子类中定义的方法与父类中的方法具有相同的方法名字、相同的参数列表、相同的返回类型（也允许子类中方法的返回类型是父类中方法返回类型的子类），对于实例方法来说这就是方法覆盖，对于静态方法来说这就是方法隐藏。

```
public class A {  
    public void m(int x, int y){  
        // do something  
    }  
    // m方法的重载  
    public void m(double x, double y){  
        // do something  
    }  
}  
  
class B extends A{  
    // m方法的重载  
    public void m(float x, float y){  
        // do something  
    }  
    // m方法的覆盖  
    public void m(int x, int y){  
        // do something else  
    }  
}
```

```

    }
    // 下面这个方法既不是重载也不是覆盖
    public int m(int x, int y){
        }
}

```

## 9.6 可见性修饰符

类访问控制符：

- public
- 包级（默认）

类成员访问控制符以及作用：

- private：只能被当前类定义的函数访问
- protected：子类、同一包中的类的函数可以访问
- public：所有类的函数可以访问
- 包级（默认）：无修饰符的成员，只能被同一包中的类访问

访问权限	本类	本包	子类	它包
public	√	√	√	√
protected	√	√	√	✗
包级（默认）	√	√	✗	✗
private	√	✗	✗	✗

Java继承时无继承控制(见继承，即都是公有继承，和C++不同)，故父类成员继承到派生类时访问权限保持不变（除了私有）。

## 9.7 变量的作用域和访问优先级

- 类的成员变量额作用域是整个类，和声明位置无关
- 如果一个成员变量的初始化依赖于另一个变量，则另一个便令必须在前面声明
- 如函数的局部变量i与类的成员变量i名称相同，那么优先访问局部变量i，成员变量i被隐藏(可通过this.i或类名.i访问)。

## 9.8 this引用

this引用指向调用某个方法的当前对象。

```

public class Foo {
    protected int i = 5;
    protected static double k = 1.0;

    void setI(int i){

```

```
    this.i = i;
}
static void setK(double k){
    Foo.k = k;
}

public static void main(String[] args) {
    Foo foo1 = new Foo();
    Foo foo2 = new Foo();

    foo1.setI(19);      // this指向f1
    foo2.setI(2);       // this指向f2
}
}
```

# Chapter 10 Thinking of OOP

---

## 10.1 软件开发过程

- 面向过程开发：一个软件系统由一系列过程构成。因而采用功能划分或者模块分解的方法进行
- 面向对象开发：一个软件系统由一系列参与活动的对象构成。因而需要建立对象模型，动态模型和功能模型
- 对象模型：描述类和类之间的关系，包括：关联、聚合、组合、依赖、继承
- UML建模语言的类图能够描述对象的组织结构和行为

## 10.2 类之间的关系描述

- 关联关系 (association)：关联关系是一种通用的二元关系，对象间通过活动发生联系。例如学生选课程，老师教授课程。
- 聚合关系 (aggregation)：是一种拥有关系，表示整体与部分之间的关系。所有者为据记者，从属对象成为被聚集者。**在聚合关系中，一个对象可以被多个聚集者拥有！**
- 组合关系 (composition)：是一种隶属关系，表示从属者强烈依赖于聚集者。**一个从属者只能被一个聚集者所拥有，聚集者负责从属者的创建和销毁！**
- 依赖关系 (dependency)：一个类 (client) 依存另一个类 (supplier) 的关系。
- 继承关系 (inheritance)：表示父类和子类之间的关系。
- 实现关系 (realization)：表示类和接口之间的关系。

# Chapter 11 继承和多态

## 11.1 类继承、子类和父类的ISA关系

### 11.1.1 基本语法

```
class Class2 extends Class1{  
    // Class Body  
}
```

Class1称为父类 (superclass) , Class2称为子类 (subclass) 。

### 11.1.2 继承特性

- 子类继承父类中可访问的数据和方法，子类也可以添加新的数据和方法
- 子类不继承父类的构造函数
- 一个类只能由一个直接父类
- 父类的成员继承到子类，访问权限不变
- 父类的私有属性在子类中不可见，但是可以通过所继承的get和set方法访问和设置。

### 11.1.3 初始化块

初始化块分为实例初始化块和静态初始化块，是java类中可以出现的第四种成员（前三种：方法，属性，构造函数）。

#### 实例初始化块(instance initialization block)

- 一个用大括号括住的语句块，直接嵌于类中，不在方法内
- 它的作用就像把它放在了类的每个构造方法的最开始位置，用于初始化对象，实例初始化块先于构造函数执行，还可以截获异常
- 一般来说如果多个构造方法共享一段代码，并且每个构造方法不会调用其他构造方法，那么可以把这段公共代码放在初始化模块中
- 一个类可以有多个初始化模块，初始化时在类中按顺序执行

```
public class Book{  
    private int id = 0;           //执行次序: 1  
    public Book(int id){         //执行次序: 4  
        this.id = id  
    }  
    {  
        //实例初始化块          //执行次序: 2  
    }  
    {  
        //实例初始化块          //执行次序: 3  
    }  
}
```

#### 静态初始化块

- 由static修饰的初始化模块
- 只能访问类的静态成员
- 在JVM的 Class Loader 将类装入内存时调用（类的装入和类的实例化是两个不同步骤，首先是将类装入内存，然后再实例化类的对象）
- 一个类可以有多个静态初始化模块，类被加载时，这些模块按照在类中的顺序执行

```
public class Book{
    private static int id = 0;      //执行次序：1
    public Book(int id){
        this.id = id
    }
    static {
        //静态初始化块          //执行次序：2
    }
    static {
        //静态初始化块          //执行次序：3
    }
}
```

### 初始化模块执行顺序

- 第一次使用类时装入类
  - 如果父类没装入则首先装入父类，这是个递归的过程，直到继承链上所有祖先类全部装入
  - 装入一个类时，类的静态数据成员和静态初始化模块按它们在类中出现的顺序执行
- 实例化类的对象
  - 首先构造父类对象，这是个递归过程，直到继承链上所有祖先类的对象构造好
  - 构造一个类的对象时，按在类中出现的顺序执行实例数据成员的初始化及实例初始化模块
  - 执行构造函数函数体

```
public class InitDemo {
    InitDemo() {
        new M();
    }

    public static void main(String[] args) {
        System.out.println("1");
        new InitDemo();
    }

    {
        System.out.println("2");
    }

    static {
        System.out.println("0");
    }
}

class N {
    N() {
        System.out.println("6");
    }
}
```

```

{
    System.out.println("5");
}

static {
    System.out.println("3");
}
}

class M extends N {
    M() {
        System.out.println("8");
    }

    {
        System.out.println("7");
    }

    static {
        System.out.println("4");
    }
}

// 输出: 0 1 2 3 4 5 6 7 8

```

## 11.2 Super关键字

- 利用super关键字可以显示调用父类的构造函数
  - 调用方法: `super(parameters);`
  - 必须是子类构造函数的第一条语句且仅一条语句
  - 如果子类构造函数中没有显示调用父类构造函数, 那么将自动调用父类不带参数的构造函数
  - 父类的构造函数在子类构造函数之前执行
- 访问父类成员 (包括静态和实例成员)
  - super不能用于静态上下文, 即静态方法和静态初始化模块中均不能使用, this也是
  - `super.dataName` 可以访问父类中非私有的属性
  - `super.method(paramaters)` 可以调用父类中的方法
  - 不能链式调用, 即不能 `super.super.data`

## 11.3 方法覆盖

子类重新定义了从父类中继承的**实例方法**称为方法覆盖 (override) , 方法覆盖有以下约束和特点:

- 父类的方法必须是可以访问的, 即私有方法不能被覆盖
- 静态方法不能被覆盖, 如果静态方法在子类中重新定义, 那么父类方法将被隐藏
- 一旦父类中的方法被子类覆盖, 同时用父类型的引用变量引用了子类对象, 这时不能通过这个父类型引用变量去访问被覆盖的父类方法, 即**这时被覆盖的父类方法不可再被发现**。
- 在子类函数中可以使用super调用被覆盖的父类方法
- 父类的变量 (实例变量、静态变量) 和静态方法在子类被重新定义, 但由于类的变量 (实例和静态) 和静态方法没有多态性, 因此通过父类型引用变量访问的一定是父类变量、静态方法(即被隐藏的可再发现)。

```

public class overrideDemo {
    public static void main(String[] args) {
        A o = new B();
        o.m();           // B's m
        o.s();           // A's s
    }
}

class A{
    public void m(){
        System.out.println("A's m");
    }

    public static void s(){
        System.out.println("A's s");
    }
}

class B extends A{
    public void m(){
        System.out.println("B's m");
    }

    public static void s(){
        System.out.println("B's s");
    }
}

```

详解：引用变量o有二个类型：声明类型A，实际运行时类型B。

判断o.s()执行的是哪个函数按照o的声明类型，因为静态函数s没有多态性，函数入口地址在编译时就确定（早期绑定），而编译时所有变量的类型都按声明类型。

判断o.m()执行的是哪个函数按照o的实际运行类型，在运行时按照o指向的实际类型B来重新计算函数入口地址（晚期绑定。多态性），因此调用的是B的m。

## 11.4 Object类

java.lang.Object类是所有类的祖先类。如果一个类在声明时没有指定父类，那么这个类的父类是Object类。它提供方法如`toString`、`equals`、`getClass`、`clone`、`finalize`，前3个为公有，后2个为保护。`getClass`为final（用于泛型和反射机制，禁止覆盖）。

### 实现equals

`equals`用于判断一个对象同另一个对象的所有成员内容是否相等。覆盖时应考虑：

- 对基本类型数值成员。直接使用`==`判断即可。
- 对引用类型变量成员。则需要对这些变量成员调用`equals`判断，不能用`==`。
- 覆盖`equals`函数，最好同时覆盖`hashCode()`方法，该方法返回对象的`hashCode`值。需要对比的时候，首先用`hashCode`去对比，如果`hashCode`不一样，则表示这两个对象肯定不相等（也就是不必再用`equals()`去再对比了），如果`hashCode`相同，此时再用`equals()`比，如果`equals()`也相同，则表示这两个对象是真的相同了，这样既能大大提高了效率也保证了对比的绝对正确性！
- 覆盖`equals`函数，首先用`instanceof`检查参数的类型是否和当前对象的类型一样。

### 实现clone

要实现一个类的clone方法

- 首先这个类需要实现Cloneable接口，否则会抛出CloneNotSupportedException异常，Cloneable接口其实就是一个标记接口，里面没有定义任何接口方法，只是用来标记一个类是否支持克隆：没有实现该接口的类不能克隆
- 公有覆盖**clone方法，即Object类里clone方法是保护的，子类覆盖这个方法时应该提升为public
- 方法里应实现深拷贝clone，Object的clone实现是浅拷贝（按成员赋值）。
- 克隆的深度：要克隆的对象可能包含基本类型数值成员或引用类型变量成员，对于基本类型数值成员使用=赋值即可，对于引用类型成员则需要进一步嵌套调用该成员的克隆方法进行赋值。

## 11.5 多态性、动态绑定和对象的强制类型转换

继承关系使一个子类可以继承父类的特征（属性和方法），并附加新特征；子类是父类的具体化，每一个子类的实例都是父类的实例，但是反过来不成立。

```
Class Student extends Person{ ...}
Person p = new Student(); //OK 父类引用可直接指向子类对象
Student s = new Person(); //error
```

### 11.5.1 多态

多态指的是通过引用变量调用实例函数时，根据所引用的实际对象的类型，执行该类型的相应实例方法，从而表现出不同的行为称为多态。通过继承时覆盖父类的实例方法实现多态。多态实现的原理：在运行时根据引用变量指向对象的实际类型，重新计算调用方法的入口地址（晚期绑定）。

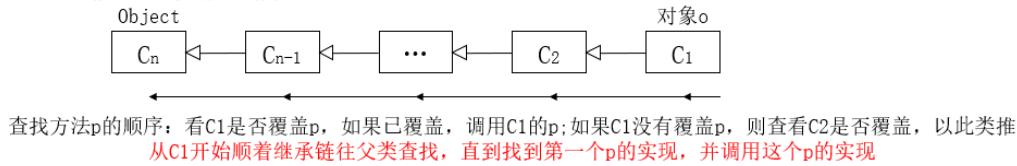
```
class Person{ void Greeting(){ System.out.println("Best wish from a person!"); }
}
class Employee extends Person{
    void Greeting(){ System.out.println("Best wish from a employee!"); }
}
class Manager extends Employee{
    void Greeting(){ System.out.println("Best wish from a manager!"); }
}

public class GreetingTest1{
    public static void main(String[] args){
        //父类引用变量可以引用本类和子类对象，p1,p2,p3的声明类型都是Person(父类型)，p2,p3
        //指向子类对象
        Person p1= new Person(),p2= new Employee(),p3= new Manager();
        p1.Greeting(); //调用Person的Greeting() ，由于实际指向对象类型是Person
        p2.Greeting(); //调用Employee的Greeting() ，由于实际指向对象类型是Employee
        p3.Greeting(); //调用Manager的Greeting() ，由于实际指向对象类型是Manager
    }
}
```

## 11.5.2 动态绑定

当调用实例方法时，由Java虚拟机动态地决定所调用的方法，称为动态绑定(dynamic binding)或者晚期绑定或者延迟绑定(lazy binding)或者多态。

假定对象 $\circ$ 是类 $C_1$ 的实例， $C_1$ 是 $C_2$ 的子类， $C_2$ 是 $C_3$ 的子类，...， $C_{n-1}$ 是 $C_n$ 的子类。也就是说， $C_n$ 是最一般的类， $C_1$ 是最具体的类。在Java中， $C_n$ 是Object类。如果调用继承链里子类型 $C_1$ 对象 $\circ$ 的方法 $p$ ，Java虚拟机按照 $C_1$ 、 $C_2$ 、...、 $C_n$ 的顺序依次查找方法 $p$ 的实现。一旦找到一个实现，将停止查找，并执行找到的第一个实现(覆盖的实例函数)。



## 11.5.3 类型转换

类型转换(type casting)可以将一个对象的类型转换成继承链中的另一种类型。

- 从子类到父类的转换是合法的，称为隐式转换。

```
Person p = new Manager(); // 将子类对象转换为父类对象
```

- 从父类到子类必须强制类型转换

```
Manager m = (Manager)p; // Manager m = p 编译会报错
```

- 从父类到子类转换必须显示转换，转换前应进行检查更加安全

```
Manager m = null;  
if(p instanceof Manager) m= (Manager)p; //安全：转换前检查
```

## 11.5.4 summary

- 重载发生在编译时(Compile time)，编译时编译器根据实参比对重载方法的形参找到最合适的方法。
- 多态发生在运行(Run time)时，运行时JVM根据变量所引用的对象的真正类型来找到最合适的方法。
- 有的书上把重载叫做“编译时多态”，或者叫“早期绑定”(早期指编译时)。
- 多态是晚期绑定(晚期指运行时)
- 绑定是指找到函数的入口地址的过程。

# Chapter 12: 异常处理和文本IO

## 12.1 异常处理概述

异常(Exception)又称为例外，是程序在运行过程中发生的非正常事件，其发生会影响程序的正常执行。当一个方法中发生错误时，将创建一个对象并将它交给运行时系统，此对象被称为异常对象(exception object)。创建异常对象并将它交给运行时系统被称为抛出一个异常(throw an exception)。

异常产生有以下原因：

- Java虚拟机同步检测到一个异常的执行条件，**间接抛出**异常，例如：
  - 表达式违反了正常语义，例如整数除0
  - 通过空引用访问实例变量或方法
  - 访问数组越界

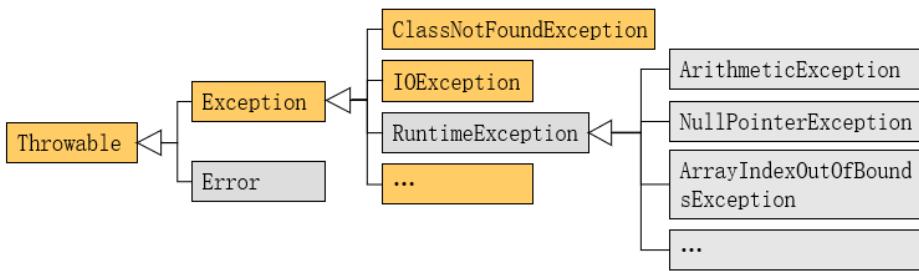
```
public class A {  
    public void m1(){ }  
    public static void main(String[] args){  
        A o = null;  
        /*  
         * 通过空引用访问实例方法，会间接地抛出异常NullPointerException  
         */  
        o.m1();  
    }  
}
```

- 通过执行**throw**语句**显示抛出**异常
  - 程序在某个条件下，用**throw**直接抛出异常

```
public class A {  
    //由于main方法里抛出的异常没有被处理，因此在main方法必须加上异常声明throws Exception  
    public static void main(String[] args) throws Exception{  
        int i = new Scanner(System.in).nextInt();  
        if(i > 10){ //假设应用逻辑要求用户输入整数不能大于10  
            throw new Exception("Input value is too big"); //显式地用throw抛出异常  
        }  
    }  
}
```

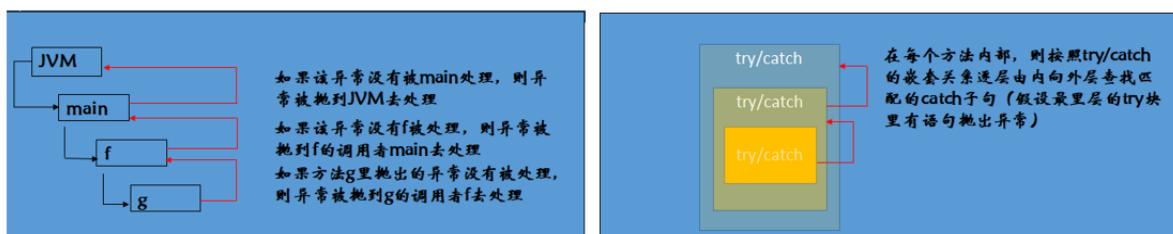
### 异常特点：

- 异常**都必须继承**Throwable的直接或间接子类。用户通过继承自定义异常
- 异常分为两类：从Exception派生的程序级错误，可由程序本身处理；从Error派生的系统级错误，程序可不用处理
- Exception的子类中又分为**必检异常**和**非必检异常**，属于RuntimeException这个分支的异常都是非必检异常。除了RuntimeException这个分支外，其他都是必检异常，即要么在函数中用**catch**捕获，要么在函数上加上异常声明。



### 运行时异常处理过程:

- 当发生异常时，运行时系统按与方法调用次序相反的次序搜索调用堆栈，寻找一个包含可处理异常的代码块的方法，这个代码块称为异常处理器(exception handler)，即try/catch语句
- 如果被抛出的异常对象与try/catch块可以处理的类型匹配，运行时系统将异常对象传递给它，这称为捕获异常(catch the exception)
- 如果运行时系统彻底搜索了调用堆栈中的所有方法，但没有找到合适的异常处理器，程序则终止



```

public class CallStack {
    public static void methodA(){
        System.out.println("in A");
        try {
            methodB();
        } catch (Exception e) {
            System.out.println(e);
        }
        System.out.println("end A");
    }

    public static void methodB(){
        System.out.println("in B");
        methodC();
        System.out.println("end B");
    }

    public static void methodC(){
        System.out.println("in C");
        int i = 10/0;
        System.out.println(i);
        System.out.println("end C");
    }

    public static void main(String[] args) {
        System.out.println("in Main");
        methodA();
        System.out.println("end Main");
    }
}

```

```
// 输出:  
// in Main  
// in A  
// in B  
// in C  
// java.lang.ArithmetricException: / by zero  
// end A  
// end Main
```

## 12.2 异常声明、抛出和捕获

### 12.2.1 异常种类

非必检异常(Unchecked Exception)是运行时异常(RuntimeException)和错误(Error)类及它们的子类, 非必检异常在方法里可不捕获异常同时方法头可不声明异常, 编译器不会报错。但该发生的异常还是要发生。

必检异常(Checked Exception), 编译器确保必检异常被捕获(通过try/catch)或声明(通过在方法头上是同throws子句声明可能抛出异常), 即要不在方法里捕获异常, 要不在方法头声明异常。

### 12.2.2 异常声明

#### 1. 由方法声明可能抛出的异常

- 如果方法不捕获其中发生的必检异常, 那么方法必须声明它可能抛出的这些异常
- 通过throws子句声明方法可能抛出的异常。throws子句由throws关键字和一个以逗号分隔的列表组成, 列表列出此方法抛出的所有异常, 即一个方法可以声明多个可能抛出的异常

```
public void myMethod() throws IOException {  
    InputStream in =  
        new FileInputStream(new File("C:\\1.txt"));  
}
```

#### 2. 抛出异常

- 情况一: 间接抛出。执行语句(如new FileInputStream(new File("C:\\1.txt"))); 或调用方法时由被调用方法抛出的异常
- 情况二: 显示直接抛出

```
int i = new Scanner(System.in).nextInt();  
if(i > 10){ //假设应用逻辑要求用户输入整数不能大于10  
    throw new Exception("Input value is too big");  
}
```

### 12.2.3 异常捕获

语法：

```
try {
    statements
} catch (ExceptionType1 id1) {
    statements1
} catch (ExceptionType2 id2) {
    statements2
} finally {
    statements3
}
// 包含catch子句, finally子句是非必要的
// 包含finally子句, catch子句是非必要的
```

- 将可能抛出异常的语句放在try块中。当try块中的语句发生异常时，异常由后面的catch块捕获处理。
- 一个try块后面可以有多个catch块。每个catch块可以处理的异常类型由异常类型参数指定。异常参数类型必须是从Throwable派生的类。
- 当try块中的语句抛出异常对象时，运行时系统将调用第一个异常对象类型与参数类型匹配的catch子句。如果被抛出的异常对象可以被合法地赋值给catch子句的参数，那么系统就认为它是匹配的（和方法调用传参一样，子类异常对象匹配父类型异常参数类型）。
- 无论try块中是否发生异常，都会执行finally块中的代码。通常用于关闭文件或释放其它系统资源。
- 处理异常时，也可以抛出新异常，或者处理完异常后继续向上（本方法调用者）抛出异常以让上层调用者知道发生什么事情：链式异常。

```
public static String read(String filePath) {
    String s = null;
    BufferedReader reader = null; //BufferedReader一次读文本文件一行
    try{
        StringBuffer buf = new StringBuffer();
        reader = new BufferedReader(new InputStreamReader(new FileInputStream(new
            File(filePath))));
        while( (s = reader.readLine()) != null){//readLine方法读取到文件末尾返回null
            buf.append(s).append("\n");
        }
        s = buf.toString().trim();
    }
    catch (FileNotFoundException e) { e.printStackTrace();}
    catch (IOException e) { e.printStackTrace();}
    finally {
        if(reader != null) {
            try { reader.close();}
            catch (IOException e) { e.printStackTrace();}
        }
    }
    return s;
}
```

new FileInputStream可能抛出  
FileNotFoundException，怎么知道的？通过FileInputStream构造函数方法头

readLine方法可能抛出 IOException。  
怎么知道的？通过readLine的方法头的  
throws声明

try块里可能抛出的二个异常分别被二个  
catch块处理

由于reader打开后，执行readLine时可能抛出异常，  
因此在finally块里关闭流是最合适的地方。注意  
close也可能抛出异常，因此还得用try/catch处理

方法read内部已经处理了所有可能发生的异常，  
因此方法首部不需要加throws声明。同时  
read方法的调用代码不需要try/catch

## 12.3 异常捕获的顺序

- 每个catch根据自己的参数类型捕获相应的类型匹配的异常
- 由于父类引用参数可接受子类对象，因此，若把Throwable作为第1个catch子句的参数，它将捕获任何类型的异常，导致后续catch没有捕获机会。
- 通常将继承链最底层的异常类型作为第1个catch子句参数，次底层异常类型作为第2个catch子句参数，以此类推。**越在前面的catch子句其异常参数类型应该越具体**。以便所有catch都有机会捕捉相应异常。
- 无论何时，throw以后的语句都不会执行。

- 无论同层catch子句是否捕获、处理本层的异常（即使在catch块里抛出或转发异常），同层的finally总是都会执行。
- 一个catch捕获到异常后，同层其他catch都不会执行，然后执行同层finally。

```

public class Main {
    static int div(int x, int y) { //各种Exception都被捕获，函数无须声明异常
        int r=0;
        try{
            //自己抛出异常对象
            if(y==0) throw new ArithmeticException();
            r=x/y;
        } catch(ArithmeticException ae) { System.out.print(ae.toString()); throw ae; }
        catch(Exception ae){//捕获各种Exception: 若是第1个catch，则后续的catch子句无机会捕获
            System.out.print(ae.toString());
        }
        finally{ r=-1; } //无论是否有异常，r=-1
        return r;
    }
    public static void main(String[ ] args) {
        try{ div(5, 0); } //调用div(5, 0)后，div函数的执行轨迹已用红色标出
        catch(Throwable ae) { //任何异常都被捕获，包括Error类型异常
            System.out.print(ae.toString());
        }
    }
}

```

处理完异常后可以继续抛出异常，交给上层调用者继续处理。注意即使这里抛出异常，同层的finally仍会执行

catch子句里抛出异常，这个异常在div方法里没有处理，但是div可以不声明异常？为什么？因为ae是非必检

因此虽然div没有异常声明，在main里调用div也用了try/catch

## 12.4 自定义异常类

自定义异常类必须继承Throwable或其子类，自定义异常类通常继承Exception及其子类，因为Exception是程序可处理的类。如果自定义异常类在父类的基础上增加了成员变量，通常需要覆盖toString函数。自定义异常类通常不必定义clone：捕获和处理异常时通常只是引用异常对象而已。

```

import java.lang.Exception;
public class ValueBeyondRangeException extends Exception{
    int value, range;
    public ValueBeyondRangeException(int v, int r){ value=v; range=r; }
    public toString(){
        return value + " beyonds " + range;
    }
}
//使用例子
int v = 1000, range = 100;
try{
    if(v > range)
        throw new ValueBeyondRangeException (v,range);
}
catch(ValueBeyondRangeException e){ System.out.println(e.toString()); }

```

## 12.5 文本IO

- 文本：非二进制文件（二进制文件参见FileInputStream、 FileOutputStream）。
- 类库：java.io.File、java.util.Scanner、java.io.PrintWriter。
- 类File：对文件和目录的抽象，包括：路径管理，文件读写状态、修改日期获取等。
- 类Scanner：从File或InputStream的读入。可按串、字节、整数、双精度、或整行等不同要求读入。
- 类PrintWriter：输出到File或OutputStream：可按串、字节、整数、双精度、或整行等不同要求输出。

```
public class Copy {
```

```
public static void main(String[ ] args) { //参数不含程序名
    if(args.length!=2){
        System.out.println("Usage: Java Copy <sourceFile> <targetFile>");
        System.exit(1);
    };
    File sF=new File(args[0]); //args[0]:源文件路径
    if(!sF.exists( )) {
        System.out.println("Source File "+args[0]+ "does not exist!");
        System.exit(2);
    };
    File tF=new File(args[1]); //args[1]:目标文件
    if(tF.exists( )) {
        System.out.println("Target File "+args[0]+ "already exist");
        System.exit(3);
    };
    try{
        Scanner input=new Scanner(sF);
        PrintWriter output=new PrintWriter(tF);
        while(input.hasNext( )) {
            String s=input.nextLine(); //读取下一行
            output.println(s); //打印这一行
        }
        input.close();
        output.close();
    }
    catch(IOException ioe){
        System.out.println(ioe.toString());
    }
}
}
```

# Chapter 19 泛型

## 19.1 基本概念

泛型 (Generic) 指可以把类型参数化，这个能力使得我们可以定义带类型参数的泛型类、泛型接口、泛型方法，随后编译器会用**唯一的具体类型替换它**；

主要优点是在**编译时检测出错误**。泛型类或方法允许用户指定可以和这些类或方法一起工作的对象类型。如果试图使用一个不相容的对象，编译器就会检测出这个错误。

Java的泛型通过擦除法实现，和C++模板生成多个实例类不同。编译时会用类型实参代替类型形参进行严格的语法检查，然后擦除类型参数、生成所有实例类型共享的唯一原始类型。这样使得泛型代码能兼容老的使用原始类型的遗留代码。

```
// 泛型类
public class Wrapper<T> {
    // do something
}

Wrapper<String> stringwrapper = new Wrapper<String>();
Wrapper<Circle> circlearrpper = new Wrapper<Circle>();
```

T是一个类型变量，它可以是Java中的任何**引用类型**，例如String, Integer, Double等。当把一个具体的类型实参传递给类型形参T时，就得到了一系列的参数化类型(Parameterized Types)，如Wrapper, Wrapper，这些参数化类型是泛型类Wrapper的实例类型。

## 19.2 Class类和Class对象

类型信息是通过Class类（类名为Class的类）的对象表示的，Java利用Class对象来执行RTTI (Run-Time Type Identification运行时类型识别)。通过运行时类型信息，程序在运行时能够检查父类引用所指的对象的实际派生类型。

每个类都有一个对应的Class对象，每当编写并编译了一个类，就会产生一个Class对象，这个对象当JVM加载这个类时就产生了。

### 19.2.1 获取Class对象

#### 1. 通过Class.forName方法获取

```
public class Person {
    public static void main(String[] args) {
        try {
            Class c1z = Class.forName("www.learnjava.garfield.ch19.Manager");
            System.out.println(c1z.getName());
            // 获取完全限定名 www.learnjava.garfield.ch19.Manager
            System.out.println(c1z.getSimpleName());
            // 获取简单名 Manager

            Class superc1z = c1z.getSuperclass();
            // 获取直接父类
            System.out.println(superc1z.getName());
        }
    }
}
```

```

        // 获取完全限定名 www.learnjava.garfield.ch19.Employee
        System.out.println(superclz.getSimpleName());
        // 获取简单名 Employee
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
}

class Employee extends Person{

}

class Manager extends Employee{
}

```

编译器是无法检查字符串"ch13.Manager"是否为一个正确的类的完全限定名，因此在运行时可能抛出异常，比如当不小心把类名写错了时。

## 2. 通过类字面量获取

```

public class Person {
    public static void main(String[] args) {
        Class clz = Manager.class;
        System.out.println(clz.getName()); // 获取完全限定名
www.learnjava.garfield.ch19.Manager
        System.out.println(clz.getSimpleName()); // 获取简单名 Manager

        Class superclz = clz.getSuperclass(); // 获取直接父类
        System.out.println(superclz.getName()); // 获取完全限定名
www.learnjava.garfield.ch19.Employee
        System.out.println(superclz.getSimpleName()); // 获取简单名 Employee

    }
}

class Employee extends Person {

}

class Manager extends Employee {
}

```

类字面量不仅可以用于类，也可用于数组(int[].class)，接口，基本类型，如int.class

相比Class.forName方法，这种方法更安全，在编译时就会被检查，因此不需要放在Try/Catch块里（见上面的标注里说明）

Class.forName会引起类的静态初始化块的执行，T.class不会引起类的静态初始化块的执行

## 3. 通过对对象获取

```

public class Person {
    public static void main(String[] args) {
        Object o = new Manager();
        Class clz = o.getClass()
        System.out.println(clz.getName()); // 获取完全限定名
www.learnjava.garfield.ch19.Manager
        System.out.println(clz.getSimpleName()); // 获取简单名 Manager

        Class superclz = clz.getSuperclass(); // 获取直接父类
        System.out.println(superclz.getName()); // 获取完全限定名
www.learnjava.garfield.ch19.Employee
        System.out.println(superclz.getSimpleName()); // 获取简单名 Employee

    }
}

class Employee extends Person {

}

class Manager extends Employee {
}

```

## 19.2.2 反射 (reflection)

利用反射，我们可以在运行时动态地创建对象，调用对象的方法。

```

public class ReflectDemo {
    public static void main(String[] args) {
        Class clz = Student.class;
        Constructor[] constructors = clz.getConstructors(); // 获得构造函数
        Method[] methods = clz.getMethods();

        try {
            //实例化对象
            //1: 如有缺省构造函数，调用Class对象的newInstance方法
            Student s1 = (Student)clz.newInstance();
            //2. 调用带参数的构造函数,
            // 到参数类型为String的构造函数对象，然后调用它的newInstance方法调用构造函数，参数为“John”。等价于：
            // Student s2 = new Student("John");
            // 但是是通过反射机制调用的
            Student s2 =
            (Student)clz.getConstructor(String.class).newInstance("John");

            // invoke method.得到方法名为setName，参数为String的方法对象m，类型是Method。
            // 然后通过m.invoke去调用该方法，第一个参数为对象，第二个参数是传递给被调方法的实参。
            // 这二条语句等价于s1.setName("Marry)，但是是通过反射去调的
            Method m = clz.getMethod("setName", String.class);
            m.invoke(s1, "Marry"); //调用s1对象的setName方法，实参"Marry"
            System.out.println(s1.toString());
            System.out.println(s2.toString());
        }
    }
}

```

```

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

}

class Student{
    private String name;

    public Student(){

    }

    public Student(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Student{" +
            "name='" + name + '\'' +
            '}';
    }
}

```

## 19.3 定义泛型类和接口

### 19.3.1 用泛型定义栈类

```

public class GenericStack<T> {
    private ArrayList<T> list = new ArrayList<>();

    //注意泛型类的构造函数不带泛型参数，连<>都不能有
    public GenericContainer(){}
}

public boolean isEmpty() {
    return list.isEmpty();
}

public int getSize() {
    return list.size();
}

public T peek() {
    return list.get(list.size() - 1);
}

```

```

    }

    public T pop() {
        return list.remove(list.size() - 1);
    }

    public void push(T t) {
        list.add(t);
    }
}

```

注意：

- `GenericStack<E>` 构造函数形式是擦除参数类型后的 `GenericStack()`, 不是 `GenericStack<>()`
- 泛型类或者泛型接口的一个实例类型，可以作为其它类的父类或者类要实现的接口，  
例如 `public class Circle implements Comparable<Circle>`

## 19.4 泛型方法

```

public class GenericMethodDemo {

    public static void main(String[] args) {
        Integer[] integers = {1,2,3,4,5};
        GenericMethodDemo.<Integer>print(integers);
    }

    public static <T> void print(T[] arr) {
        for (T t : arr) {
            System.out.println(t.toString());
        }
    }
}

```

调用泛型方法，将实际类型放于<>之中方法名之前；也可以不显式指定实际类型，而直接给实参调用，如 `print(integers); print(strings);` 由编译器自动发现实际类型。

声明泛型方法，将类型参数置于返回类型之前，方法的类型参数可以作为形参类型，方法返回类型，也可以用在方法体内其他类型可以用的地方。

## 19.5 原始类型

没有指定具体类型实参的泛型类和泛型接口称为原始类型（raw type）。如：

```
GenericStack stack = new GenericStack(); 等价于 GenericStatck<Object> stack = new
GenericStack<Object>();
```

这种**不带类型参数的泛型类或泛型接口称为原始类型**。使用原始类型可以向后兼容Java的早期版本。如Comparable类型。**尽量不要用**。

```
//从JDK1.5开始，Comparable就是泛型接口Comparable<T>的原始类型(raw type)
public class Max {
    public static Comparable findMax(Comparable o1, Comparable o2){
        return (o1.compareTo(o2) > 0)?o1:o2;
    }
}
```

上例中， Comparable o1 和 Comparable o2 都是原始类型声明，但是，**原始类型是不安全的**。  
`Max.findMax("welcome", 123)`；编译通过，但会引起运行时错误。安全的办法是使用泛型，现在将 `findMax` 方法改成泛型方法。

```
public class Max {
    public static <E extends Comparable<E>> E findMax(E o1, E o2){
        return (o1.compareTo(o2) > 0)?o1:o2;
    }
}
// E extends Comparable<E>>指定类型E必须实现Comparable接口，而且接口比较对象类型必须是E
// 注意：在指定受限的类型参数时，不管是继承父类还是实现接口，都用extends
public class Circle implements Comparable<Circle> {...}

Max.findMax(new Circle(), new Circle(10.0));
//编译上面这条语句时，编译器会自动发现findMax的类型实参为Circle，用Circle替换E
```

## 19.6 通配泛型

### 19.6.1 通配泛型

通配泛型有三种形式：

- ？，非受限通配，等价于 ? extends Object，注意，`GenericStack<?>` 不是原始类型，`GenericStack` 是原始类型
- ? extends T, 受限通配，表示 T 或者 T 的子类，上界通配符，T 定义了类型上限
- ? super T, 下限通配，表示 T 或者 T 的父类，下界通配符，T 定义了类型下限

### 19.6.2 协变性问题

数组的协变性：如果类 A 是类 B 的父类，那么 A[] 就是 B[] 的父类。

```
class Fruit{}
class Apple extends Fruit{}
class Jonathan extends Apple{} //一种苹果
class Orange extends Fruit{}

//由于数组的协变性，可以把Apple[]类型的引用赋值给Fruit[]类型的引用
Fruit[] fruits = new Apple[10];
fruits[0] = new Apple();
fruits[1] = new Jonathan(); // Jonathan是Apple的子类

try{
    //下面语句fruits的声明类型是Fruit因此编译通过，但运行时将Fruit转型为Apple错误
    //数组是在运行时才去判断数组元素的类型约束；
    //而泛型正好相反，在运行时，泛型的类型信息是会被擦除的，编译的时候去检查类型约束
}
```

```
fruits[2] = new Fruit(); //运行时抛出异常 java.lang.ArrayStoreException, 这是数组  
协变性导致的问题  
}catch(Exception e){  
    System.out.println(e);  
}
```

为了解决数组协变性导致的问题，Java编译器规定泛型容器（任何泛型类）没有协变性。

```
ArrayList<Fruit> list = new ArrayList<Apple>(); //编译错误  
//Type mismatch: cannot convert from ArrayList<Apple> to ArrayList<Fruit>
```

我们在谈论容器的类型，而不是容器持有对象的类型

A是B父类型，但泛型类(比如容器)，`ArrayList<A>`不是`ArrayList <B>`的父类型。因此，上面语句报错。

为什么数组有协变性而泛型没有协变性：

数组具有协变性是因此在运行时才去判断数组元素的类型约束（前一页PPT例子），这将导致有时发生运行时错误，抛出异常`java.lang.ArrayStoreException`。这个功能在Java中是一个公认的“瑕疵”

泛型没有协变性：泛型设计者认为**与其在运行失败，不如在编译时就失败**（禁止泛型的协变性就是为了杜绝数组协变性带来的问题，即如果泛型有协变性，面临可协变的数组一样的问题）——静态类型语言（Java,C++）的全部意义在于代码运行前找出错误。Python, JavaScript之类的语言是动态类型语言。

但有时希望像数组一样，一个父类型容器引用变量指向子类型容器，这时要使用**通配符**

- **采用上界通配泛型 ? extends**

```
ArrayList<? extends Fruit> list = new ArrayList<Apple>(); //左边类型是右边类型的父类型
```

- **上面语句编译通过，但是这样的list不能加入任何东西。下面语句都会编译出错**

```
list.add(new Apple()); list.add(new Fruit()); //编译都报错  
//可加入null  
list.add(null);
```

```
//但是从这个list取对象没有问题，编译时都解释成Fruit，运行时可以是具体的类型如Apple (多态性)  
Fruit f = list.get(0);
```

- 因为`ArrayList<? extends Fruit>`意味着该list集中存放的都是Fruit的子类型（包括Fruit自身），Fruit的子类型可能有很多，但list只能存放其中的某一种类型。编译器只能知道元素类型的上限是Fruit，而无法知道list引用会指向什么具体的ArrayList，可以是`ArrayList<Apple>`,也可能是`ArrayList<Jonathan>`，为了安全，Java泛型只能将其设计成不能添加元素。
- 虽然不能添加元素，但从里面获取元素的类型都是Fruit类型（编译时）
- 因此带`<? extends>`类型通配符的泛型类**不能往里存内容（不能set），只能读取（只能get）**
- 那这样声明的容器类型有什么意义？它的意义是作为一个只读（只从里面取对象）的容器

## ● 采用下界通配泛型？ super

```
//采用下界通配符 ? super T 的泛型类引用，可以指向所有以T及其父类型为类型参数的实例类型
ArrayList<? super Fruit> list = new ArrayList<Fruit>(); //这时new后的Fruit可以省略
ArrayList<? super Fruit> list = new ArrayList<Object>(); //允许， Object是Fruit父类
ArrayList<? super Fruit> list = new ArrayList<Apple>(); //但是不能指向Fruit子类的容器
```

## ● 可以向list里面添加T及T的子类对象

```
list.add(new Fruit());           //OK
list.add(new Apple());          //OK
list.add(new Jonathan());       //OK
list.add(new Orange());         //OK
//list.add(new Object()); //添加Fruit父类则编译器禁止， 报错
```

## ● 但是从list里get数据只能被编译器解释成Object

```
Object o1 = list.get(0); //OK
Fruit o2 = list.get(0); //报错， Object不能赋给Fruit， 需要强制类型转换，
```

## ● 因此这种泛型类和采用? extends的泛型类正好相反：只能存数据，获取数据至少部分失效（编译器解释成Object）

### ● ? extends 和? super的理解

//现在看看通配泛型 ? extends，注意右边的new ArrayList的类型参数必须是Fruit的子类型  
//? extends Fruit指定了类型上限，因此下面的都成立：

```
ArrayList<? extends Fruit> list1 = new ArrayList<Fruit>(); //=号右边，如果是Fruit，可以不写，等价于new ArrayList<>();
ArrayList<? extends Fruit> list2 = new ArrayList<Apple>(); //=号右边，如果是Fruit的子类，则必须写
ArrayList<? extends Fruit> list3 = new ArrayList<Jonathan>(); //=号右边，如果是Fruit的子类，则必须写
ArrayList<? extends Fruit> list4 = new ArrayList<Orange>(); //=号右边，如果是Fruit的子类，则必须写
```

ArrayList<? extends Fruit> list可指向ArrayList<Fruit>|ArrayList<Apple>|ArrayList<Jonathan>| ArrayList<Orange>|...

一个ArrayList<Fruit>容器可以加入Fruit、Apple、Jonathan、Orange，  
一个ArrayList<Apple>容器可以加入Apple、Jonathan，  
一个ArrayList<Orange>容器可以加入Orange，  
假如当ArrayList<? extends Fruit> list为方法形参时，如果方法内部调list.add，  
由于编译时，编译器无法知道ArrayList<? extends Fruit>类型的引用变量会指向哪一个具体容器类型，编译器无法知道该怎么处理add。  
例如当add的对象类型是Orange，如果list指向ArrayList<Apple>，加不进去。但如果list指向为ArrayList<Orange>，就可以加进去。  
**为了安全，编译器干脆禁止ArrayList<? extends Fruit>类型的list添加元素。**  
**但从list里get元素，都解释成Fruit类型！**

### ● ? extends 和? super的理解

```
//? super Fruit指定了类型下限，因此下面二行都成立
ArrayList<? super Fruit> list1 = new ArrayList<Fruit>(); //=-号右边，这时Fruit可以省略，等价于new ArrayList<>();
ArrayList<? super Fruit> list2 = new ArrayList<Object>(); //允许。=-号右边，如果是Fruit的父类，必须写出类型
//ArrayList<? super Fruit> list3 = new ArrayList<Apple>(); //但是不能指向Fruit子类的容器
```

因此ArrayList<? super Fruit> list引用可以指向ArrayList<Fruit>|Fruit父类型的容器如ArrayList<Object>。  
当ArrayList<? super Fruit> list为方法形参时，编译器知道list指向的具体容器的类型参数至少是Fruit。当向list里add对象o时，分析几种可能的情况：

1 o是Fruit及其子类类型，这里面又分两种情况

- 1.1 ArrayList<? super Fruit> list实际指向ArrayList<Fruit>，可以加入
- 1.2 ArrayList<? super Fruit> list实际指向ArrayList<Object>，可以加入

2 o是Fruit的父类型如Object，这里面又分两种情况

- 2.1 ArrayList<? super Fruit> list实际指向ArrayList<Fruit>，这时编译器不允许加入，Object不能转型为Fruit
- 2.2 ArrayList<? super Fruit> list实际指向ArrayList<Object>，可以加入

综合以上四种情况，可以看到，只要对象o的类型是Fruit及其子类型，这时将对象o加入list一定是安全的（1.1, 1.2）；

如果对象是Fruit父类型，则不允许加入最安全（因为可能出现2.1的情况）。由于? super Fruit规定了list元素类型的下限，因此取元素时编译器只能全部解释成Object

```
list1.add(new Fruit()); list1.add(new Apple()); list1.add(new Jonathan());//只要加入Fruit及其子类对象都OK
//list1.add(new Object()); //添加Fruit父类则编译器禁止， 报错
```

取对象时都必须解释成Object类型。因此我们说带<? super>通配符的泛型类的get方法至少是部分失效

```
Object o1 = list.get(0);
//Fruit o2 = list.get(0); //报错， Object不能赋给Fruit， 需要强制类型转换，但是引入泛型就是想去掉强制类型转换
```

### 19.6.3 使用原则

? extends 和 ? super 的使用原则为PECS: Producer Extends, Consumer Super

- Producer Extends: 如果需要一个只读泛型类, 用来Produce T, 那么用? extends T
- Consumer Super: 如果需要一个只写泛型类, 用来Consume T, 那么用? super T

## 19.7 泛型擦除和对泛型的限制

### 19.7.1 泛型擦除

泛型是用类型擦除 (type erasure) 方法实现的。泛型的作用就是使得编译器在编译时通过类型参数来检测代码的类型匹配性。**当编译通过, 意味着代码里的类型都是匹配的。因此, 所有的类型参数使命完成而全部被擦除。**因此, 泛型信息(类型参数)在运行时是不可用的, 这种方法使得泛型代码向后兼容使用原始代码的遗留代码。

泛型存在于编译时, 当编译器认为泛型类型是安全的, 就会将其转化为原始类型。这时(a)所示的源代码编译后变成(b)所示的代码。**注意在(b)里, 由于list.get(0)返回的对象运行时类型一定是String, 因此强制类型转换一定是安全的。**

```
ArrayList<String> list = new ArrayList<>();  
list.add("Oklahoma");  
String state = list.get(0);
```

a)

```
ArrayList list = new ArrayList();  
list.add("Oklahoma");  
String state = (String)(list.get(0));
```

b)

当编译泛型类、接口和方法时, 会用Object代替非受限类型参数E。<E extends Object>

```
public static <E> void print(E[] list) {  
    for (int i = 0; i < list.length; i++)  
        System.out.print(list[i] + " ");  
    System.out.println();  
}
```

a)

```
public static void print(Object[] list) {  
    for (int i = 0; i < list.length; i++)  
        System.out.print(list[i] + " ");  
    System.out.println();  
}
```

b)

如果一个泛型的参数类型是受限的, 编译器会用该受限类型来替换它。

```
public static <E extends GeometricObject>  
    boolean equalArea(  
        E object1,  
        E object2) {  
    return object1.getArea() ==  
        object2.getArea();  
}
```

a)

```
public static  
    boolean equalArea(  
        GeometricObject object1,  
        GeometricObject object2) {  
    return object1.getArea() ==  
        object2.getArea();  
}
```

b)

### 19.7.2 泛型限制

- 不能使用new E(); //只能想办法得到E的类型实参的Class信息, 再newInstance(...)

不能用泛型的类型参数创建实例, 如: E object = new E(); //错误

- 不能使用new E[]

不能用泛型的类型参数创建数组, 如: E[] element = new E[capacity]; //错误

**注意：泛型类型参数在运行时不可用！！new是运行时发生的，因此new后面一定不能出现类型形参E，运行时类型参数早没了**

- 强制类型转换可以用类型形参E，通过类型转换实现无法确保运行时类型转换是否成功

`E[] element = (E[])new Object[capacity];` 编译可通过(所谓编译通过就是指编译时uncheck，至于运行时是否出错，那是程序员自己的责任)

- 静态上下文中不允许使用泛型的类型参数。由于泛型类的所有实例类型都共享相同的运行时类，所以泛型类的静态变量和方法都被它的所有实例类型所共享。**因此，在静态方法、数据域或者初始化语句中，使用泛型的参数类型是非法的。

```
public class Test<E> {  
    public static void m(E o1) { // Illegal  
    }  
  
    public static E o1; // Illegal  
  
    static {  
        E o2; // Illegal  
    }  
}
```

Test<String>和Test<Integer>这两个实例类型共享同一个运行时类型，如果静态上下文可以使用类型参数E，会导致矛盾：  
m方法的形参类型到底是String还是Integer？

- 异常类（Exception类）是不能泛型的。因为JVM在运行时捕获异常必须检查类型，但是在运行时，泛型都已经擦除了！

泛型限制举例：

```
public class GenericOneDimensionArrayUncheck<T> { //实现一维数组的泛型包装类。不可能  
实现泛型数组  
    private T[] elements; //T[]类型数组存放元素  
    public GenericOneDimensionArrayUncheck(int size){  
        //new Object[] 强制类型转换。强制类型转换就是uncheck，就是强烈要求编译器把=右边的类  
型解释成T[]  
        elements = (T[])new Object[size]; //注意：在运行时，elements引用变量指向的是  
object[]  
    }  
    //这里value的类型是T，这点非常重要，保证了放进去的元素类型必须是T及子类型。否则编译报错  
    public void put(T value,int index){ elements[index] = value; }  
    public T get(int index){ return elements[index]; } //elements声明类型就是T[], 因  
此类型一致  
    public T[] getElements() {return elements;} //这个方法非常危险，编译没问题  
    public static void main(String[] args){  
        GenericOneDimensionArrayUncheck<String> strArray = new  
            GenericOneDimensionArrayUncheck<>(10);  
        strArray.put("Hello",0);  
        // strArray.put(new Fruit(),0); //不是String对象放不进去  
        String s = strArray.get(0); //strArray.get(0)返回对象的运行时类型一定是  
String，由put保证的  
        //但是下面的语句抛出运行时异常：java.lang.ClassCastException  
        //因为运行时，elements引用变量指向的是Object[], 无法转成String[]  
        String[] a = strArray.getElements(); //返回内部数组，但为String[]类型  
    }  
}
```

```

public class GenericOneDimensionArray<T> {
    private T[] elements = null; //T[]类型

    public GenericOneDimensionArray(Class<? extends T> c1z,int size){
        elements = (T[])Array.newInstance(c1z,size);
        // 这里第一个参数是Class<? extends T> c1z, 表示一个T类型及其子类的Class对象。
        // 通过Class对象, 可以通过反射创建运行时类型为T[]的数组。
        // 但是Array.newInstance方法返回的是Object, 因此需要强制类型转换。
        // 但这里的强制类型转换是安全的, 因为创建的数组的运行时类型就是T[]
        // Array.newInstance(数组元素类型的Class对象, size)
        // 通过反射机制创建运行时类型为T[]的数组
    }

}

//get, put等其他方法省略

public T[] getElements(){ return elements; }

public static void main(String[] args){
    GenericOneDimensionArray<String> stringArray =
        new GenericOneDimensionArray(String.class,10);
    String[] a = stringArray.getElements(); //这里不会抛出运行时异常了
    //      a[0] = new Fruit(); //不是String类型的对象, 编译报错
    a[1] = "Hello";
}
}

```

### 19.7.3 实现带泛型参数的对象工厂

```

public class ObjectFactory<T> {
    private Class<T> type; // 保存要创建的对象的类型信息
    public ObjectFactory(Class<T> type) {
        this.type = type;
    }
    public T create(){
        T o = null;
        try {
            o= type.newInstance();
        } catch (InstantiationException | IllegalAccessException e) {
            e.printStackTrace();
        }
        return o;
    }
}

public class TestFactory {
    public static void main(String[] args) {
        //首先创建一个负责生产Car的对象工厂, 传进去需要创建对象的类的Class信息
        ObjectFactory<Car> carFactory = new ObjectFactory<Car>(Car.class);
        Car o = carFactory.create(); //由对象工厂负责产生car对象
        System.out.println(carFactory.create().toString());
    }
}

```

```
    }

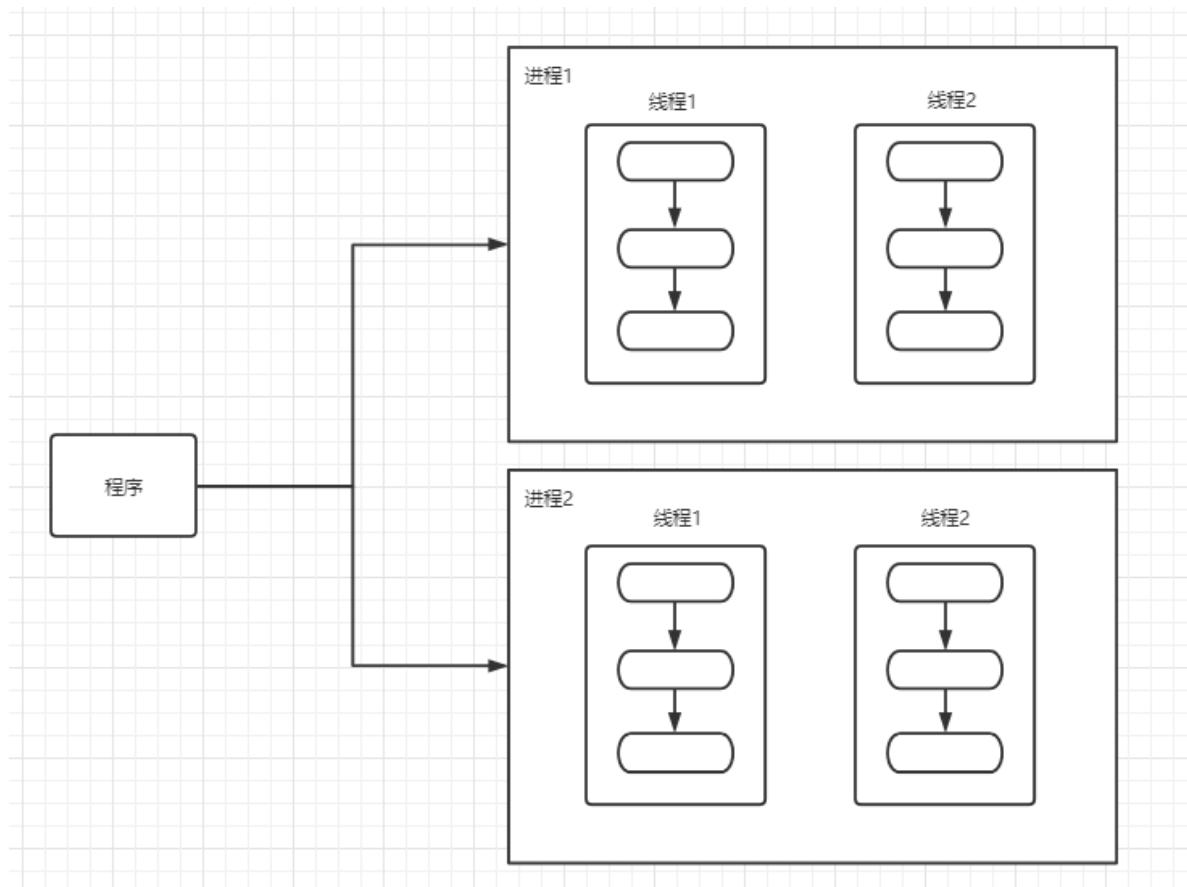
class Car {
    private String s = null;
    public Car() {
        s = "Car";
    }
    public String toString() {
        return s;
    }
}
```

# Chapter 30 多线程

## 30.1 线程的概念

### 30.1.1 程序、进程和线程

- 程序：一个静态的对象，内含指令和数据的文件，存储在磁盘或其他存储设备中
- 进程：一个动态的对象，是程序的一个执行过程，存在于系统内存中。一个进程对应于一个程序
- 线程：运行于某个进程中，用于完成某个具体任务的顺序控制流程，**程序中完成一个任务的有始有终的执行流，都有一个执行的起点，经过一系列指令后到达终点。**



不同进程的内存空间是隔离的，因此进程1中的变量i与进程2中的变量i属于不同的内存空间。进程切换和进程间通信开销大。进程间交换数据只能通过：共享内存、管道、消息队列、Socket通信等机制

一个进程里的线程切换开销小的多，因为它们位于同一内存空间里。线程1、2线程位于同一内存空间使得线程之间数据交换非常容易。变量i可以被线程1、2访问（但要考虑同步）。因此线程又叫轻量级进程

**当一个进程被创建，自动地创建了一个主线程。因此，一个进程至少有一个主线程。**

### 30.1.2 线程作用

- 一个进程的多个子线程可以**并发运行**。
- 多线程可以使程序 反应更快、交互性更强、执行效率更高。
- 应用：**Server端程序， GUI程序**
  - Server端的程序，都是需要启动多个线程来处理大量来自客户端的请求

- GUI程序：GUI线程：处理UI消息循环，如鼠标消息、键盘消息；Worker线程：后台的数据处理工作，比如打印文件，大数据量的运算

## 30.2 Runnable接口和Thread线程类

### 30.2.1 通过实现Runnable接口创建线程

具体步骤：

1. 实现Runnable接口，实现唯一的接口方法run
2. 创建实现Runnable接口的类的具体对象
3. 利用Thread类的构造函数创建线程对象
4. 通过线程对象的start方法启动线程

代码举例：

```
public class RunnableDemo implements Runnable{
    public RunnableDemo(){
        // constructor function
    }

    @Override
    public void run() {
        // do something
    }

    public static void main(String[] args) {
        // 创建一个实现了Runnable接口的实例
        Runnable task = new RunnableDemo();
        // 创建一个线程，注意new Thread()中的参数必须是实现了Runnable接口的实例
        Thread thread1 = new Thread(task);
        // 启动线程，会执行task的run方法
        thread1.start();
    }
}
```

注意：任何线程只能启动依次，多次调用产生IllegalThreadStateException异常

### 30.2.2 通过继承Thread类创建线程

具体步骤：

1. 定义Thread类的扩展类，在扩展类中重写run方法
2. 通过扩展类创建线程对象
3. 通过线程对象的start方法启动线程

代码举例：

```
public class ThreadDemo {
    public static void main(String[] args) {
        Thread thread1 = new PrintCharDemo2('a');
        Thread thread2 = new PrintCharDemo2('b');
        thread1.start();
        thread2.start();
    }
}
```

```

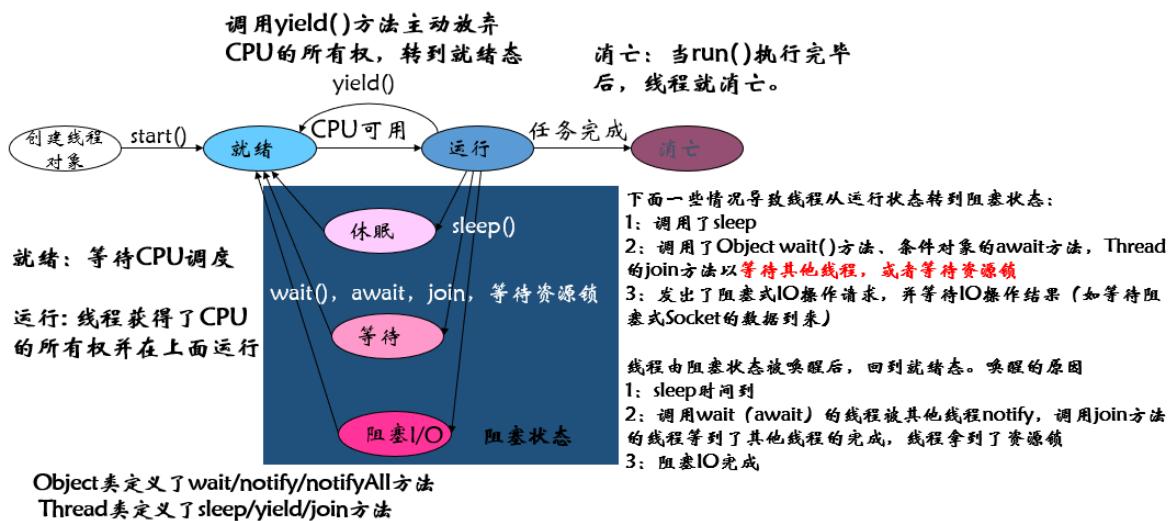
    }

// 扩展类继承了Thread类
class PrintCharDemo2 extends Thread{
    private char printedChar;
    public PrintCharDemo2(char a){
        this.printedChar = a;
    }

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println(this.printedChar);
        }
    }
}

```

### 30.2.3 线程的状态切换



### 30.2.3 Thread类常见的方法

- `start();`开始一个线程，进入Ready状态，如无其他线程等待，则立即Run进入running状态
- `isAlive();`获取线程当前是否在运行
- `setPriority(p:int);`为该线程指定优先值p:1~10
- `join();`等待线程结束
- `sleep(millis:long);`让当前线程休眠若干ms，监视器自动恢复其运行
- `yield();`将线程从running变为ready，允许其他线程执行（自己也可能立即执行）
- `interrupt();`中断该线程

yield用法：

```

public class YieldDemo extends Thread{
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println('a');
            Thread.yield(); // 挂起进入ready, 给其他进程调度机会
        }
    }
}

```

sleep用法:

```

public class SleepDemo extends Thread{
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println('a');
            if (i>50){
                // 处于阻塞状态(如在睡眠, 在wait, 在执行阻塞式IO)的线程,
                // 如果被其他线程打断(即处于阻塞的线程的interrupt方法被其它线程调用),
                // 会抛出InterruptedException, 是一个必检异常
                try {
                    Thread.sleep(5);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}

```

join用法:

```

public class JoinDemo extends Thread{
    public static void main(String[] args) {
        Thread threadA = new Thread(new PrintChar('a'));
        Thread threadB = new Thread(new PrintChar('b'));
        threadA.start();           // 启动线程A
        try {
            threadA.join();      // 主线程被阻塞, 等待线程A执行完毕
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        threadB.start();         // 主线程被唤醒, 启动线程B
    }
}

class PrintChar implements Runnable{
    private char printedChar;

    public PrintChar(char printedChar) {
        this.printedChar = printedChar;
    }
}

```

```

    }

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println(printedChar);
        }
    }

}

```

join方法的作用：在A线程中调用了B线程（对象）的join()方法时，表示A线程放弃控制权（被阻塞了），只有当B线程执行完毕时，A线程才被唤醒继续执行。

程序在main线程中调用A线程（对象）的join方法时，main线程放弃cpu控制权（被阻塞），直到线程A执行完毕，main线程被唤醒执行threadB.start();

运行结果是全部a打印完才开始打印b

### 30.3 线程池

线程池适合大量线程任务的并发执行。线程池通过有效管理线程、“复用”线程来提高性能。从JDK 1.5 开始使用Executor接口（执行器）来执行线程池中的任务，Executor的子接口ExecutorService管理和控制任务。我们只需要把实例化好的线程任务对象（Runnable接口实例）交给执行器Executor就可以了。Thread由线程池内部来创建和维护。

代码举例：

```

public class ThreadPoolDemo {
    public static void main(String[] args) {
        // 创建一个线程池，最大容量为3个
        ExecutorService es = Executors.newFixedThreadPool(3);
        // 提交Runnable的任务给线程池
        es.execute(new PrintString("Thread A running"));
        es.execute(new PrintString("Thread B running"));
        es.execute(new PrintString("Thread C running"));
        // 关闭线程池
        es.shutdown();
    }
}

class PrintString implements Runnable{
    private String s;

    public PrintString(String s) {
        this.s = s;
    }

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println(s);
        }
    }
}

```

区分任务和线程的区别：

任务：实现了Runnable接口的类的实例，任务的逻辑由run方法实现

线程：是Thread类的实例，是任务的运行载体，任务必须通过线程来运行

`Thread thread = new Thread(task)` 中thread是线程，task是任务。

## 30.4 线程同步

如果多个线程对一个对象进行操作，如果操作的顺序不一致，会导致结果不一致，因此需要同步线程。

临界区：可能被多个线程同时进入的程序的一部分区域，可以是方法，也可以是语句块。因此我们需要对临界区进行同步，保证任何时候只有一个线程进入临界区。

### 30.4.1 synchronized关键字实现同步

#### synchronized同步方法

**synchronized**是通过加锁来实现方法同步的：一个线程要进入同步方法，首先拿到锁，进入方法后立刻上锁，导致其他要进入这个方法的线程被阻塞（等待锁）。

对于synchronized实例方法，是对该方法的对象加锁

对于synchronized静态方法，是对该方法的类加锁

当进入方法的线程执行完方法后，锁被释放，会唤醒等待这把锁的其他线程

基本语法：`public synchronized void deposit(double amount)`

#### synchronized同步语句块

基本语法：`synchronized (expr) { statements; }`

表示式expr结果必须是对一个对象的引用，因此可以通过对任何对象加锁来同步语句块。

- 如果expr指向的对象没有被加锁，则第一个执行到同步块的线程对该对象加锁，线程执行该语句块，然后解锁；
- 如果expr指向的对象已经加了锁，则执行到同步块的其它线程将被阻塞
- expr指向的对象解锁后，所有等待该对象锁的线程都被唤醒

同步语句块允许同步方法中的部分代码，而不必是整个方法，增强了程序的并发能力

任何同步的实例方法都可以转换为同步语句块

```
public synchronized void xMethod() {  
    // method body  
}
```

```
public void xMethod() {  
    synchronized (this) {  
        // method body  
    }  
}
```

### 30.4.2 加锁来实现同步

采用synchronized关键字的同步要隐式地在对象实例或类上加锁，粒度较大影响性能，JDK 1.5 可以显式地加锁，能够在更小的粒度上进行线程同步，一个锁是一个Lock接口的实例，类ReentrantLock是Lock的一个具体实现：可重入的锁

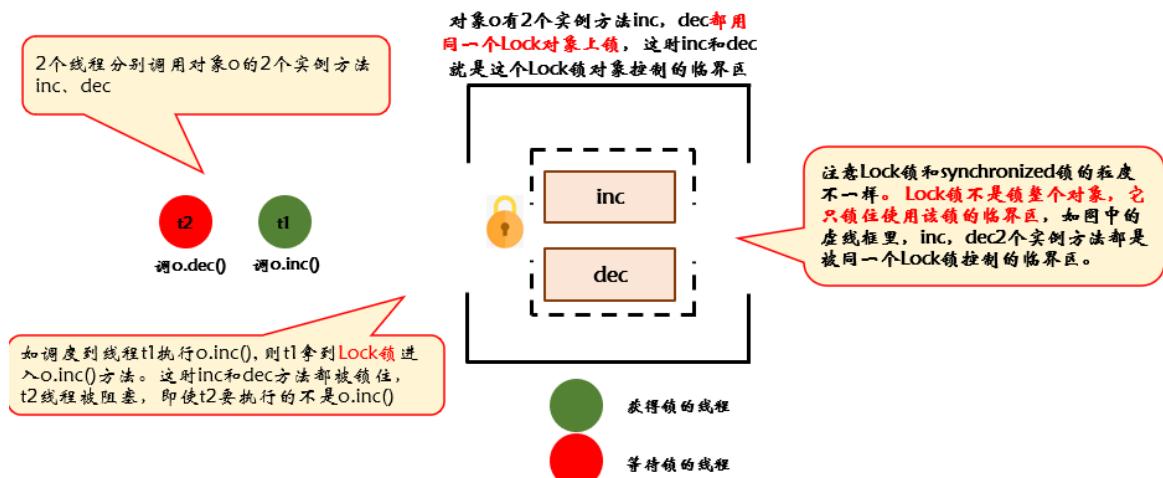
伪代码：

```
void methodA(){
    lock.lock(); // 获取锁
    methodB();
    lock.unlock() // 释放锁
}

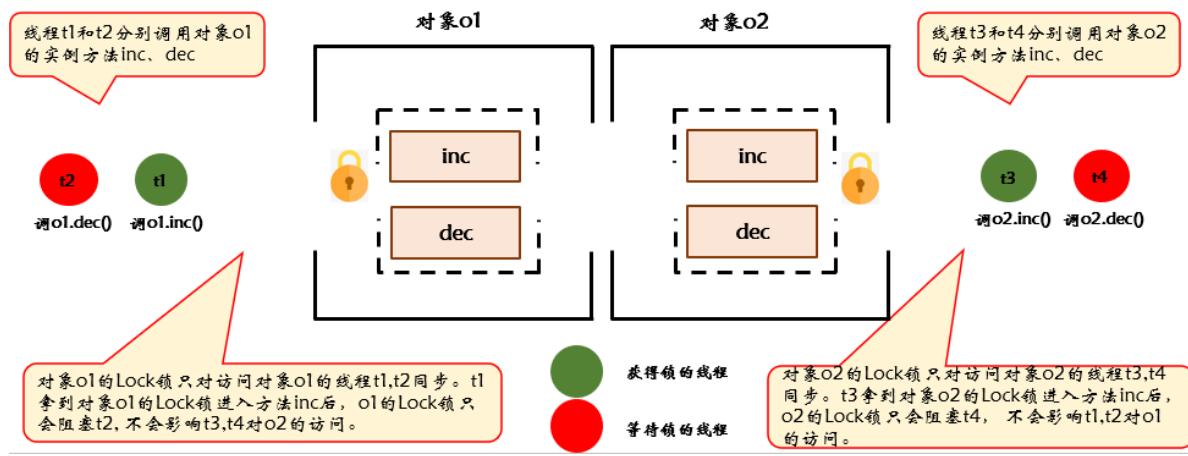
void methodB(){
    lock.lock(); // 再次获取该锁
    // 其他业务
    lock.unlock(); // 释放锁
}
```

### 30.4.3 锁的总结

- 情景1：假设一个类有多个用synchronized修饰的同步实例方法，如果多个线程访问这个类的同一个对象，当一个线程获得了该对象锁进入到其中一个同步方法时，**这把锁会锁住这个对象所有的同步实例方法**
- 情景2：假设一个类有多个用synchronized修饰的同步实例方法，如果多个线程访问这个类的不同对象，那么**不同对象的synchronized锁不一样，每个对象的锁只能对访问该对象的线程同步**
- 情景3：如果采用Lock锁进行同步，**一旦Lock锁被一个线程获得，那么被这把锁控制的所有临界区都被上锁**，这时所有其他访问这些临界区的线程都被阻塞。



- 情景4：如果一个类采用Lock锁对临界区上锁，而且这个Lock锁也是该类的实例成员，**那么这个类的二个实例的Lock锁就是不同的锁**，下面的动画演示了这种场景：对象o1的Lock锁和对象o2的Lock锁是不同的锁对象。



如果采用synchronized关键字对类 A 的实例方法进行同步控制，这时等价于synchronized(this){ }

一旦一个线程进入类A的对象o的synchronized实例方法，对象o被加锁，对象o所有的synchronized实例方法都被锁住，从而阻塞了要访问对象o的synchronized实例方法的线程，但是与访问A类其它对象的线程无关

如果采用synchronized关键字对类 A 的静态方法进行同步控制，这时等价于synchronized(A.class){ }。一旦一个线程进入A的一个静态同步方法，A所有的静态同步方法都被锁（这个锁是类级别的锁），这个锁对所有访问该类静态同步方法的线程有效，不管这些线程是通过类名访问静态同步方法还是通过不同的对象访问静态同步方法。

如果通过Lock对象进行同步，首先看Lock对象对哪些临界区上锁，一旦Lock锁被一个线程获得，那么被这把锁控制的所有临界区都被上锁（如场景3）；另外要区分Lock对象本身是否是不同的：不同的Lock对象能阻塞的线程是不一样的（如场景4）。

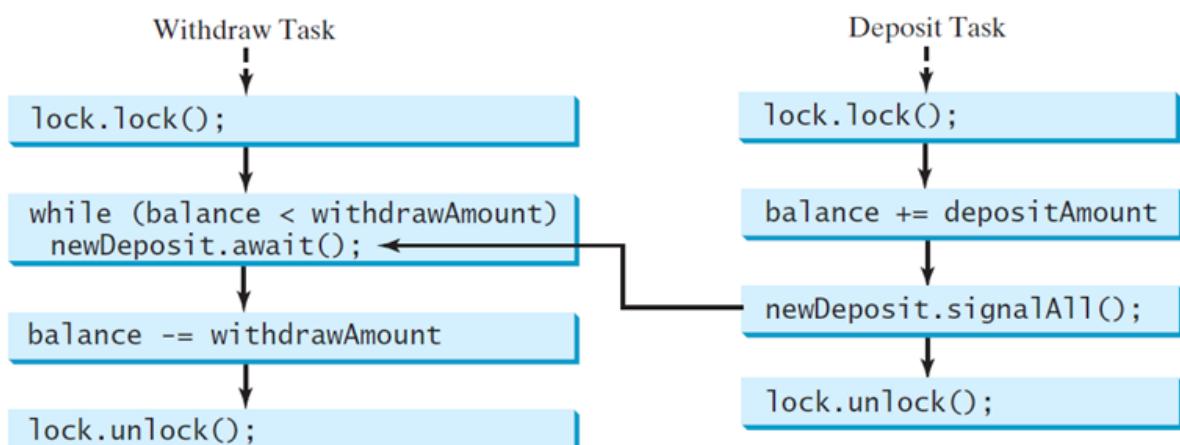
#### 30.4.4 线程合作

线程之间有资源竞争，synchronized和Lock锁这些同步机制解决的是资源竞争问题

线程之间还有相互协作的问题

假设创建并启动两个任务线程：

- 存款线程用来向账户中存款
- 提款线程从同一账户中提款
- 当提款的数额大于账户的当前余额时，提款线程必须等待存款线程往账户里存钱
- 如果存款线程存入一笔资金，必须通知提款线程重新尝试提款，如果余额仍未达到提款的数额，提款线程必须继续等待新的存款



线程之间的相互协作：可通过Condition对象的await/signal/signalAll来完成

- Condition(条件)对象是通过调用Lock实例的newCondition( )方法而创建的对象
  - Condition对象可以用于协调线程之间的交互(使用条件实现线程间通信)
  - 一旦创建了条件对象condition，就可以通过调用condition.await()使当前线程进入等待状态，
  - 其它线程通过同一个条件对象调用signal和signalAll()方法来唤醒等待的线程，从而实现线程之间的相互协作

## 线程合作举例：

```
        condition.signalAll();  
    } finally {  
    lock.unlock();  
}
```

Thread1	Thread2	Balance
deposit5		5
	Withdraw 2	3
	Withdraw 2	1
deposit7		8
	Waiting for deposit	
deposit3		11
	Withdraw 10	1
	Waiting for deposit	
deposit4		5
	Waiting for deposit	
deposit6		11
	Waiting for deposit	

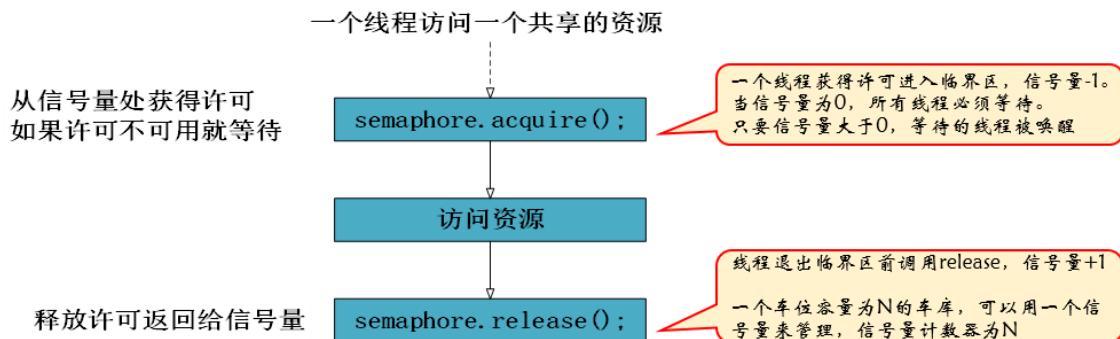
## 30.5 信号量

信号量用来限制访问一个共享资源的线程数，是一个有计数器的锁，访问资源之前，线程必须从信号量获取许可，访问完资源之后，该线程必须将许可返回给信号量

为了创建信号量，必须确定许可的数量（计数器最大值），同时可选用公平策略

任务通过调用信号量的acquire()方法来获得许可，信号量中可用许可的总数减1

任务通过调用信号量的release()方法来释放许可，信号量中可用许可的总数加1



```

import java.util.concurrent.Semaphore;
// An inner class for account
private static class Account {
    // Create a semaphore
    private static Semaphore semaphore = new Semaphore(1);
    private int balance = 0;
    public int getBalance() {return balance;}

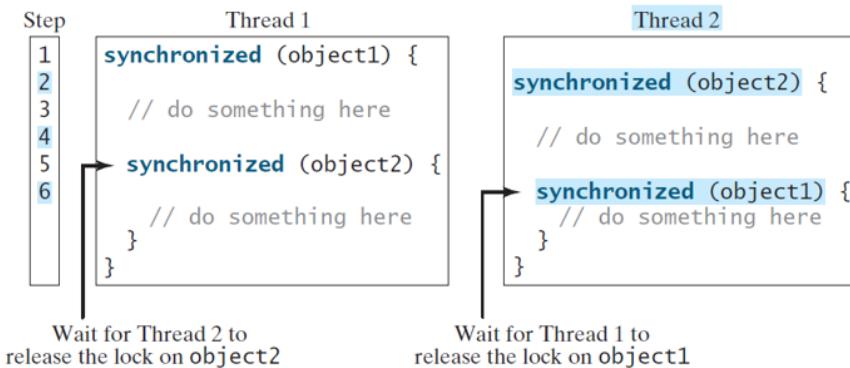
    public void deposit(int amount) {
        try {
            semaphore.acquire();
            int newBalance=balance+amount
            Thread.sleep(5);
            balance=new Balance;
        }
        finally {
            semaphore.release();
        }
    }
}

```

```
    }
}
}
```

## 30.6 避免死锁

- 死锁：如图所示，两个线程形成死锁



- 避免死锁：可以采用正确的资源排序来避免死锁

- 给每一个需要上锁的对象指定一个顺序
- 确保每个线程都按这个顺序来获取锁

线程2必须先获取object1上的锁，然后才能获取Object2上的锁

## 30.7 同步合集

■ Java集合框架 包括：List、Set、Map接口及其具体子类，都不是线程安全的。

■ 集合框架中的类不是线程安全的，可通过为访问集合的代码临界区加锁或者同步等方式来保护集合中的数据

■ Collections类提供6个静态方法来将集合转成同步版本（即线程安全的版本）

■ 这些同步版本的类都是线程安全的，但是迭代器不是，因此使用迭代器时必须同步：  
synchronized(要迭代的集合对象){//迭代}

java.util.Collections

+synchronizedCollection(c:Collection) :Collection
+synchronizedList(list>List) :List
+synchronizedMap(m:Map) :Map
+synchronizedSet(s:Set) :Set
+synchronizedSortedMap(s:SortedMap) :SortedMap
+synchronizedSortedSet(s:SortedSet) :SortedSet

从一个给定的合集返回一个同步集合
从一个给定的线性表返回一个同步线性表
从一个给定的映射表返回一个同步映射表
从一个给定的集合返回一个同步集合
从一个给定的排序映射表返回一个同步排序映射表
从一个给定的排序集合返回一个同步排序集合