[TOC]

第2章 基本程序设计

2.1 编写简单的程序

略

2.2 从控制台读取输入

2.2.1 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

2.2.2 Scanner类（java.util.Scanner）

```
Scanner scanner = new Scanner(System.in);
//构造函数Scanner的参数类型也可为java.io.File
//这是Scanner就从文件而不是标准输入流读取数据
double d = scanner.nextDouble( );
```

方法：
nextByte( )、nextShort( )、nextInt( )
nextLong( )、nextFloat( )、nextDouble( )
next( ) 读入一个字符串
例：

```
package  test;   //类必须在一个包中
import java.util.Scanner;
public class TestScanner {
	public static void main(String[ ] args) {
		// Create a scanner
		Scanner scanner = new Scanner(System.in);

		// Prompt the user to enter an integer
		System.out.print("Enter an integer: ");
		int intValue = scanner.nextInt();
		System.out.println("You entered the integer: " + intValue);

		// Prompt the user to enter a double value
		System.out.print("Enter a double value: ");
		double doubleValue = scanner.nextDouble();
		System.out.println("You entered the double value: " + doubleValue);

		System.out.print("Enter a string without space: ");
		String string = scanner.next();
		System.out.println("You entered the string: " + string);
	}
}
```

2.3 标识符，变量，常量

2.3.1 标识符

Java中使用标识符(identifier)来命名变量、常量、方法、类、包等实体。

标识符命名规则

标识符是由**字母、数字、下划线(_)、*美元符号($)**组成的字符序列。*
标识符必须以字母、下划线(_)、美元符号($)开头。不能以数字开头。
标识符不能是保留字。
标识符不能为true、false或null等事实上的保留字（参见英文维基网）。
标识符可以为任意长度，但编译通常只接受前128字符。
例如：$2, area, radius, showMessageDialog是合法的标识符；2A, d+4是非法的标识符

Java保留字：

类别	关键字	说明
访问修饰符	`public`, `protected`, `private`	控制类、方法、变量的访问权限。
非访问修饰符	`static`, `final`, `abstract`, `transient`, `volatile`, `synchronized`	修饰类、方法、变量的行为特性（如静态、不可变、抽象等）。
数据类型	`byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean`	定义基本数据类型。
流程控制	`if`, `else`, `switch`, `case`, `default`, `while`, `do`, `for`, `break`, `continue`, `return`	控制程序执行流程。
异常处理	`try`, `catch`, `finally`, `throw`, `throws`	处理程序运行时异常。
类和对象	`class`, `interface`, `extends`, `implements`, `new`, `instanceof`, `super`, `this`	定义类、接口、继承关系和对象操作。

包和导入	`package`, `import`	管理代码组织和依赖。
其他	`void`, `null`, `true`, `false`, `enum`, `strictfp`, `native`, `const`（未使用）, `goto`（未使用）	特殊用途或保留未使用的关键字。

修饰符组合示例

修饰符组合	应用场景	示例
`public static`	修饰类方法或变量	`public static void main(String[] args)`
`private final`	修饰不可变的实例变量	`private final int MAX_VALUE = 100;`
`abstract protected`	修饰抽象方法（需子类实现）	`protected abstract void calculate();`

注意事项

1. **未使用的关键字：**
    - `const` 和 `goto` 是 Java 保留但未实际使用的关键字。
2. **严格浮点：**
    - `strictfp` 确保浮点运算在不同平台上结果一致。
3. **本地方法：**
    - `native` 修饰的方法由本地代码（如 C/C++）实现。

2.3.2 变量和常量

略

2.4 赋值语句和赋值表达式

2.4.1 赋值语句

```
int m = 5;
m += m = 3; // 等价于 m = m + (m = 3)
System.out.println(m); // 输出 8

int x = 1;
int y = x = 2 + x; // 操作数从左到右求值，但赋值从右向左
System.out.println(y); // 输出 3
```

```
int x = 1;
int y = (x = 2) + x; // 操作数从左到右求值，但赋值从右向左
System.out.println(y); // 输出 4
```

2.4.2 同时完成声明和初始化

语法

datatype variable = expression;

例如：

```
int x = 1;   //某些变量在申明时必须同时初始化：final int m=0;
int x = 1, y = 2;
```

函数里的局部变量在使用前必须赋值。

```
int x, y;     //若是类的成员变量，x, y有默认值=0
y = x + 1; //局部变量无默认值则错error
```

2.5 JAVA 数据类型

略

2.6编程风格和Java常见错误类型

略

第3章 选择

第4章 数学函数、字符和字符串

4.1 常用数学函数

Java 提供了丰富的数学函数，这些函数主要集中在 `java.lang.Math` 类中。以下是一些常见的数学函数及其用法：

1. 基本数学运算

- `Math.abs(x)`：返回 `x` 的绝对值。
- `Math.max(x, y)`：返回 `x` 和 `y` 中的较大值。
- `Math.min(x, y)`：返回 `x` 和 `y` 中的较小值。

2. 幂和根

- `Math.pow(x, y)`：返回 `x` 的 `y` 次方。
- `Math.sqrt(x)`：返回 `x` 的平方根。
- `Math.cbrt(x)`：返回 `x` 的立方根。

3. 三角函数

- `Math.sin(x)`：返回 `x` 的正弦值（`x` 为弧度）。

```
double sinValue = Math.sin(Math.PI / 2); // 结果为 1.0
```

- `Math.cos(x)`：返回 `x` 的余弦值（`x` 为弧度）。
- `Math.tan(x)`：返回 `x` 的正切值（`x` 为弧度）。
- `Math.toRadians(x)`：将角度转换为弧度。

```
double radians = Math.toRadians(180); // 结果为 3.141592653589793
```

- `Math.toDegrees(x)`：将弧度转换为角度。

```
double degrees = Math.toDegrees(Math.PI); // 结果为 180.0
```

4. 对数函数

- `Math.log(x)`：返回 `x` 的自然对数（以 `e` 为底）。

```
double logValue = Math.log(Math.E); // 结果为 1.0
```

- `Math.log10(x)`：返回 `x` 的常用对数（以 `10` 为底）。

```
double log10Value = Math.log10(100); // 结果为 2.0
```

5. 随机数

- `Math.random()`：返回一个 `[0.0, 1.0)` 之间的随机浮点数。

```
double randomValue = Math.random(); // 例如 0.123456789
```

- 一般地

```
a+(int)(Math.random( )*b)                    //返回[a, a+b)
a+(int)(Math.random( )*（b+1）)              //返回[a, a+b]
```

6. 舍入函数

- `Math.round(x)`：返回 `x` 四舍五入后的整数。
- `Math.ceil(x)`：返回大于或等于 `x` 的最小整数。
- `Math.floor(x)`：返回小于或等于 `x` 的最大整数。

7. 常量

- `Math.PI`：表示圆周率 π，约等于 `3.141592653589793`。
- `Math.E`：表示自然对数的底数 e，约等于 `2.718281828459045`。

4.2 字符数据类型和操作

4.2.1 Unicode 和 ASCII 码

- Java 对字符采用 16 位 **Unicode** 编码，因此 `char` 类型的大小为 **2 个字节**。
- 16 位的 Unicode 以 `\u` 开头的 4 位十六进制数表示，范围从 `\u0000` 到 `\uffff`，不能少写位数。
- 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 letter = 'A';
char letter = '\u0041';  // 等价，\u 后面必须写满 4 位十六进制数
```

4.2.2 字符的运算

- `++` 和 `--` 运算符也可以用在 `char` 类型数据上（因为 `char` 自动转整数），运算结果为该字符之后或之前的字符，例如：

```
char ch = 'a';
System.out.println(++ch);  // 显示 b
```

4.2.3 特殊字符的转义

- 和 C++ 一样，采用反斜杠 `\` 后面加上一个字符或一些数字位组成转义序列，一个转义序列被当做一个字符。
- 常见的转义字符：`\n`（换行）、`\t`（制表符）、`\b`（退格）、`\r`（回车）、`\f`（换页）、`\`（反斜杠）、`'`（单引号）、`"`（双引号）。
- 例如，打印带引号的信息：

```
System.out.println("He said "Java is fun"");
```

4.2.4 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。例如：

```
char ch = (char) 0xAB0041;  // 0xAB0041 是 int 字面量，要赋值给
char，必须强制类型转换
System.out.println(ch);  // 显示 A
```

- 将浮点数转换成 `char` 时，先把浮点数转成 `int` 型，然后将整数转换成 `char`：

```
char ch = (char) 65.25;
System.out.println(ch);  // 显示 A
```

- 当一个 `char` 型转换成数值型时，这个字符的 Unicode 码就被转换成某种特定数据类型：

```
int i = 'A';  // 不用强制类型转换
System.out.println(i);  // 显示 65
```

- 如果转换结果适用于目标变量（不会有精度损失），可以采用隐式转换；否则必须强制类型转换：

```
int i = 'A';
byte b = (byte) '\uFFF4';  // 取低 8 位二进制数 F4 赋值给 b
```

- 所有数值运算符都可以用在 `char` 型操作数上：
    - 如果另一个操作数是数值，那么 `char` 型操作数就自动转换为数值。
    - 如果另一个操作数是字符串，那么 `char` 型操作数会自动转换成字符串再和另一个操作数字符串相连。

```
int i = '2' + '3';  // 0x32 和 0x33
System.out.println(i);  // i 为 50 + 51 = 101
int j = 2 + 'a';  // j = 2 + 97 = 99
System.out.println(j + " is the Unicode of " + (char) j);  // 99 is the
Unicode of c
```

4.2.5 字符的比较和测试：`Character` 类

- 两个字符可以通过关系运算符进行比较，通过字符的 Unicode 值进行比较。
- Java 为每个基本类型实现了对应的包装类，`char` 类型的包装类是 `Character` 类。注意包装类对象为引用类型，不是值类型。
- `Character` 类的作用：
    - 将 `char` 类型的数据封装成对象。
    - 包含处理字符的方法和常量。
- 常用方法（均为静态方法）：
    - `isDigit`：判断一个字符是否是数字。
    - `isLetter`：判断一个字符是否是字母。
    - `isLetterOrDigit`：判断一个字符是否是字母或数字。
    - `isLowerCase`：判断一个字符是否是小写。
    - `isUpperCase`：判断一个字符是否是大写。
    - `toLowerCase`：将一个字符转换成小写。
    - `toUpperCase`：将一个字符转换成大写。

```
package hust.cs.javacourse.ch3;

public class CharacterTest {
    public static void main(String[] args) {
        System.out.println("Character.isDigit('1') is: " + Character.isDigit('1'));
        System.out.println("Character.isLetter('a') is: " + Character.isLetter('a'));
        System.out.println("Character.isLetterOrDigit('+') is: " +
Character.isLetterOrDigit('+'));
```

```
        System.out.println("Character.isUpperCase('A') is: " +
Character.isUpperCase('A'));
    }
}
```

输出：

```
Character.isDigit('1') is: true
Character.isLetter('a') is: true
Character.isLetterOrDigit('+') is: false
Character.isUpperCase('A') is: true
```

4.3 字符串类型

4.3.1 `String` 类

- `String` 类是 `final` 类，不能被继承。
- `java.lang.String` 表示一个固定长度的字符序列，实例化后其内容不能修改。

4.3.2 字符串的构造

- 从字面值创建字符串：

```
String newString = new String("Welcome to Java");
```

- 由于字符串经常使用，Java 提供了创建字符串的简写形式：

```
String message = "Welcome to Java";
```

- 字符串常量池（String Pool）：
  - 由于字符串是不可变的，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常量池中。
  - 使用 `intern()` 方法返回规范化字符串：

```
String s = new String("Welcome").intern();
```

4.3.3 字符串的比较

- `equals` 方法用于比较两个字符串是否包含相同的内容：

```
String s1 = "Hello";
String s2 = "Hello";
System.out.println(s1.equals(s2));  // true
```

- `equalsIgnoreCase` 忽略大小写比较内容是否相同。

4.3.4 字符串的比较（续）

- `compareTo` 方法用于比较两个字符串的大小，返回值为第一个不同字符的 Unicode 差值：

```
String s1 = "abc";
String s2 = "abe";
System.out.println(s1.compareTo(s2));  // -2
```

- `startsWith` 判断字符串是否以某个前缀开始：

```
String s = "Welcome to Java";
System.out.println(s.startsWith("Welcome"));  // true
```

- `endsWith` 判断字符串是否以某个后缀结束：

```
System.out.println(s.endsWith("Java"));  // true
```

4.3.5 字符串长度和获取单个字符

- `length()` 方法获取字符串的长度：

```
String s = "Hello";
System.out.println(s.length());  // 5
```

- `charAt(index)` 方法获取指定位置的字符：

```
System.out.println(s.charAt(0));  // 'H'
```

4.3.6 连接字符串

- 使用 `concat` 方法连接字符串：

```
String s1 = "Hello";
String s2 = "World";
String s3 = s1.concat(s2);  // "HelloWorld"
```

- 使用 `+` 运算符连接字符串：

```
String s4 = s1 + " " + s2;  // "Hello World"
```

4.3.7 截取子串

- `substring(beginIndex, endIndex)` 截取子串：

```
String s = "Welcome to Java";
System.out.println(s.substring(0, 7));  // "Welcome"
```

- `substring(beginIndex)` 从指定位置截取到字符串末尾：

```
System.out.println(s.substring(11));  // "Java"
```

4.3.8 字符串转换

- `toLowerCase` 将字符串转换为小写：

```
System.out.println(s.toLowerCase());  // "welcome to java"
```

- `toUpperCase` 将字符串转换为大写：

```
System.out.println(s.toUpperCase());  // "WELCOME TO JAVA"
```

- `trim` 删除字符串两端的空格：

```
String s = "  Hello  ";
System.out.println(s.trim());  // "Hello"
```

- `replace` 替换字符或字符串：

```
System.out.println(s.replace("Java", "Python"));  // "Welcome to Python"
```

4.3.9 查找字符或字符串

- `indexOf` 返回字符或字符串首次出现的位置：

```
System.out.println(s.indexOf('o'));  // 4
```

- `lastIndexOf` 返回字符或字符串最后一次出现的位置：

```
System.out.println(s.lastIndexOf('a'));  // 14
```

4.3.10 字符数组和字符串间的转换

- `toCharArray` 将字符串转换为字符数组：

```
char[] charArray = s.toCharArray();
```

- 将字符数组转换为字符串：

```
String s = new String(new char[]{'J', 'a', 'v', 'a'});
```

4.3.11 基本数据类型和字符串间的转换

- `valueOf` 将基本数据类型转换为字符串：

```
String s1 = String.valueOf(1.0);  // "1.0"
String s2 = String.valueOf(true);  // "true"
```

- 将字符串转换为基本类型：

```
double d = Double.parseDouble("3.14");
int i = Integer.parseInt("123");
boolean b = Boolean.parseBoolean("true");
```

4.4 可变字符串类：`StringBuilder` 和 `StringBuffer`

4.4.1 `StringBuilder` 和 `StringBuffer`

- `StringBuilder` 和 `StringBuffer` 用于处理可变内容的字符串。
- `StringBuffer` 是线程安全的，`StringBuilder` 是非线程安全的，但性能更高。

4.4.2 常用方法

- `append` 在字符串末尾追加数据：

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" World");  // "Hello World"
```

- `insert` 在指定位置插入数据：

```
sb.insert(5, " to");  // "Hello to World"
```

- `delete` 删除指定范围的字符：

```
sb.delete(5, 8);  // "Hello World"
```

- `reverse` 翻转字符串：

```
sb.reverse();  // "dlroW olleH"
```

- `toString` 返回字符串：

```
String s = sb.toString();
```

4.5 格式化控制台输出

4.5.1 `System.out.printf`

- JDK 1.5 开始提供了格式化控制台输出方法：

```
System.out.printf("格式：%d, %s", 99, "abc");  // 格式：99, abc
```

4.5.2 格式描述符

- 格式描述符：`%[argument_index$][flags][width][.precision]conversion`。
- 常见转换符：
    - `%b`：布尔值
    - `%c`：字符
    - `%d`：十进制整数
    - `%f`：浮点数
    - `%s`：字符串

4.5.3 示例

```
public class TestPrintf {
    public static void main(String[] args) {
        System.out.printf("boolean: %6b\n", false);
        System.out.printf("character: %4c\n", 'a');
        System.out.printf("integer: %6d, %6d\n", 100, 200);
        System.out.printf("double: %7.2f\n", 12.345);
        System.out.printf("String: %7s\n", "hello");
    }
}
```

第5章 循环

5.1 while 循环

5.1.1 语法

```
while (loop-continuation-condition)
    statement or block
```

5.1.2 说明

- `loop-continuation-condition` 是一个布尔表达式，如果为 `true`，则执行循环体；否则退出循环。
- 循环体可以是单条语句或一个代码块。

5.1.3 示例

```
int i = 0;
while (i < 5) {
    System.out.println("i = " + i);
    i++;
}
```

5.2 do-while 循环

5.2.1 语法

```
do
    statement or block
while (loop-continuation-condition);
```

5.2.2 说明

- `do-while` 循环先执行循环体，再检查 `loop-continuation-condition`。如果为 `true`，则继续循环；否则退出循环。
- 循环体至少会执行一次。

5.2.3 示例

```
int i = 0;
do {
    System.out.println("i = " + i);
    i++;
} while (i < 5);
```

5.3 for 循环

5.3.1 语法

```
for (initial-action; loop-continuation-condition; action-after-iteration)
    statement or block
```

5.3.2 说明

- `initial-action`：循环开始前执行一次，通常用于初始化循环变量。
- `loop-continuation-condition`：每次迭代前检查，如果为 `true`，则继续循环；否则退出循环。
- `action-after-iteration`：每次迭代后执行，通常用于更新循环变量。

5.3.3 示例

```
for (int i = 0; i < 5; i++) {
    System.out.println("i = " + i);
}
```

5.3.4 多表达式

`for` 循环头中的每个部分可以是零个或多个以逗号分隔的表达式。

```
for (int i = 0, j = 0; i + j < 10; i++, j++) {
    System.out.println("Welcome to Java!");
}
```

5.3.5 省略条件

如果 `for` 循环中的 `loop-continuation-condition` 被省略，则隐含为 `true`，即无限循环。

```
for (;;) {                    // 等价于 while(true) {
    // do something                 // do something
}                             // }
```

5.4 break 和 continue

5.4.1 break

```
break;
```

说明

- 立即结束包含它的最内层的循环。

5.4.2 continue

```
continue;
```

说明

- 结束当前迭代，程序控制转到循环体的结尾，继续下一次迭代。

5.4.3 示例

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        break;  // 当 i == 5 时，退出循环
    }
    if (i % 2 == 0) {
```

```
        continue;  // 当 i 为偶数时，跳过本次循环的剩余部分
    }
    System.out.println("i = " + i);
}
```

5.5 增强的 for 循环

JDK 1.5 引入了增强的 `for` 循环，可以不用下标就可以依次访问数组元素。

5.5.1 语法

```
for (elementType value : arrayRefVar) {
    // 使用 value
}
```

5.5.2 示例

```
int[] myList = {1, 2, 3, 4, 5};
int sum = 0;

// 传统 for 循环
for (int i = 0; i < myList.length; i++) {
    sum += myList[i];
}

// 增强的 for 循环
for (int value : myList) {
    sum += value;
}
```

5.5.3 说明

- `elementType` 是数组元素的类型。
- `arrayRefVar` 是数组的引用变量。
- `value` 是当前迭代的数组元素。

第6章 方法

6.1 方法的定义

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

6.1.1 语法

方法定义采用 BNF 范式（Backus-Naur Form，巴科斯范式）来描述。

6.1.2 方法签名

方法签名（Method Signature）指方法名称 +
形参列表（不含返回类型）。一个类中不能包含方法签名相同的多个方法。

6.1.3 形参和实参

方法头中声明的变量称为形参（Formal Parameter）。当调用方法时，可向形参传递一个值，这个值称为实参（Actual Parameter / Argument）。形参可以使用 `final` 进行修饰，表示方法内部不允许修改该参数（类似 C++ 的 `const`）。

- 形参不允许有默认值，最后一个可为变长参数（可用 `...` 或数组定义，参见第7章数组）。
- 方法里不允许定义 `static` 局部变量。

6.1.4 返回值

方法可以有一个返回值（Return Value）。如果方法没有返回值，返回值类型为 `void`。构造函数没有返回值（不能加 `void`）。

6.1.5 示例

```
package hust.cs.javacourse.ch6;

public class FinalParameterTest {
    public static void m(int i, final int j) {
        i = 10;  // i 可以被重新赋值
        // j = 20;  // 方法体里 final int j 不能被重新赋值
    }
}
```

6.2 调用方法

声明方法只给出方法的定义。要执行方法，必须调用（Call/Invoke）方法。

6.2.1 返回值处理

如果方法有返回值，通常将方法调用作为一个值来处理（可放在一个表达式里）。

```
int large = max(3, 4) * 2;
System.out.println(max(3, 4));
```

如果方法没有返回值，方法调用必须是一条语句。

```
System.out.println("Welcome to Java!");
```

6.2.2 控制权转移

当调用方法时，程序控制权从调用者转移至被调用的方法。当执行 `return` 语句或到达方法结尾时，程序控制权转移至调用者。

6.2.3 方法分类

- **实例方法**：必须用对象名调用（对象名：指向对象的引用变量名）。
- **静态方法**：可用类名调用，也可用对象名调用，提倡用 `类名.方法名` 调用，如 `Math.random()`。

6.2.4 调用方式

- 调用当前类中的静态方法：可直接用 `方法名`，也可用 `类名.方法名`（推荐）。
- 调用当前类中的实例方法：可用 `方法名` 或 `this.方法名` 调用（推荐）。
- 调用其它类中的静态方法：用 `类名.方法名` 或 `对象名.方法名`。
- 调用其它类的实例方法：必须用 `对象名.方法名`。

示例

```
package hust.cs.javacourse.ch6;

public class A {
    public static void staticMethodOfA1() { }
    public void instancMethodOfA1() { }
    public void instancMethodOfA2() {
        // 调用实例方法必须通过对象引用
        instancMethodOfA1();        // 调用当前类的另一个实例方法，实际上和下面语句等价
        this.instancMethodOfA1();   // 推荐用 this. 调用当前类的另一个实例方法，this
就是指向当前对象的引用

        // 调用静态方法
        A.staticMethodOfA1();       // 推荐通过类名调用静态方法
        staticMethodOfA1();         // 如果调用当前类的静态方法，类名可以省略
        B.staticMethodOfB();        // 调用另外一个类的静态方法必须用类名

        // 调用另外一个类的实例方法必须通过指向另外一个类的对象的引用
        new B().instancMethodOfB();
    }
}

package hust.cs.javacourse.ch6;

public class B {
    public static void staticMethodOfB() { }
    public void instancMethodOfB() { }
}
```

方法调用示例

```
public class TestMax {
    public static void main(String[] args) {
        int i = 5;
        int j = 2;
        int k = max(i, j);
        System.out.println("The maximum between " + i + " and " + j + " is " + k);
    }

    public static int max(int num1, int num2) {
        int result;
        result = (num1 > num2) ? num1 : num2;
        return result;
```

```
    }
}
```

6.2.5 调用堆栈

每当调用一个方法时，系统将该方法参数、局部变量存储在一个内存区域中，这个内存区域称为调用堆栈（Call Stack）。当方法结束返回到调用者时，系统自动释放相应的调用栈。

6.3 方法的参数传递

如果方法声明中包含形参，调用方法时，必须提供实参。

6.3.1 实参要求

- 实参的类型必须与形参的类型兼容：如子类实参可传递给父类形参。
- 实参顺序必须与形参的顺序一致。

示例

```
public static void nPrintln(String message, int n) {
    for (int i = 0; i < n; i++)
        System.out.println(message);
}

nPrintln("Hello", 3); // 正确
nPrintln(3, "Hello"); // 错误
```

6.3.2 参数传递方式

- **基本数据类型**：实参值的副本被传递给方法的形参。方法内部对形参的修改不影响实参值。（Call by Value）
- **对象类型**：参数是引用调用（Call by Reference）。

传递值参示例

```
public class TestPassByValue {
    public static void main(String[] args) {
        int num1 = 1;
        int num2 = 2;
        System.out.println("调用 swap 方法之前：num1 = " + num1 + "，num2 = " + num2);

        swap(num1, num2);
        System.out.println("调用 swap 方法之后：num1 = " + num1 + "，num2 = " + num2);
    }

    public static void swap(int n1, int n2) {
        System.out.println("\t在 swap 方法内：");
        System.out.println("\t\t交换之前：n1 = " + n1 + "，n2 = " + n2);

        int temp = n1;
```

```
        n1 = n2;
        n2 = temp;

        System.out.println("\t\t交换之后：n1 = " + n1 + "，n2 = " + n2);
    }
}
```

6.4 方法的重载

方法重载（Overloading）是指方法名称相同，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（同名的方法可以重载多个版本）。

形参列表不同

- 参数个数不同，或
- 至少一个参数类型不同

调用规则

当调用方法时，Java
编译器会根据实参的个数和类型寻找最合适的方法进行调用。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错误，称为歧义调用（Ambiguous Invocation）。

示例

```
public class TestMethodOverloading {
    /** Return the max between two int values */
    public static int max(int num1, int num2) {
        return (num1 > num2) ? num1 : num2;
    }

    /** Return the max between two double values */
    public static double max(double num1, double num2) {
        return (num1 > num2) ? num1 : num2;
    }

    /** Return the max among three double values */
    public static double max(double num1, double num2, double num3) {
        return max(max(num1, num2), num3);
    }
}

public class TestMethodOverloading {
    /** Main method */
    public static void main(String[] args) {
        // Invoke the max method with int parameters
        System.out.println("The maximum between 3 and 4 is " + max(3, 4)); // 调用
max(int num1, int num2)

        // Invoke the max method with the double parameters
        System.out.println("The maximum between 3.0 and 5.4 is " + max(3.0, 5.4)); //
max(double num1, double num2)

        // Invoke the max method with three double parameters
```

```
        System.out.println("The maximum between 3.0, 5.4, and 10.14 is " + max(3.0,
5.4, 10.14));
    }
}
```

歧义示例

```
public class AmbiguousOverloading {
    public static void main(String[] args) {
        // System.out.println(max(1, 2));  // 该调用产生歧义
        // 以下任一函数的参数都相容（都能自动转换），编译无法确定用哪个函数
    }

    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;
    }
}
```

6.5 方法局部变量的作用域

方法内部声明的变量称为局部变量（Local Variable）。方法的形参等同于局部变量。

作用域

局部变量的作用域（Scope）指程序中可以使用该变量的部分。局部变量的作用域从它的声明处开始，直到包含该变量的程序块 `{}`
结束。局部变量在使用前必须先赋值。局部变量的生命期和其作用域相同，因为 `{}`
结束时，局部变量出栈。

变量声明规则

- 在方法中，可以在不同的非嵌套程序块中以相同的名称多次声明局部变量。
- 但不能在嵌套的块中以相同的名称多次声明局部变量。
- 在 `for` 语句的初始动作部分声明的变量，作用域是整个循环体。在 `for` 语句循环体中声明的变量，作用域从变量声明开始到循环体结束。

示例

```
public class TestLocalVariable {
    public static void method1() {
        int x = 1; int y = 1;
        for (int i = 1; i < 10; i++) {
            x += i;
        }
        for (int i = 1; i < 10; i++) { // 正确：两个循环未嵌套，二个 for 语句的 i
互不影响
            y += i;
        }
    }
```

```
    // 错误，变量 i 在嵌套的语句块中声明：不能在嵌套块里声明同名的局部变量
    public static void method2() {
        int i = 1;
        int sum = 0;
        // for (int i = 1; i < 10; i++) { //
        //     sum += i;
        // }
    }
}
```

第7章 数组（涵盖教材第7,8章）

7.1 数组的基础知识

什么是数组

数组(array)是相同类型变量集合(这里的集合不是JDK的Collection）。数组类型的变量是引用相同类型变量集合的引用变量。凡使用new后，内存单元都初始化为`0`（值）或`null`（引用）。
数组元素本身也可以是引用变量。多维数组只是数组的数组，故数组元素也可能是引用类型变量。

声明一维数组引用变量

任何实例化的数组都是Object的子类。数组引用变量声明语法：

```
datatype[] arrayRefVar;     //提倡的写法：类型在前，[]在后
```

例如：

```
double[] myList;   //这时myList为null
```

或者

```
datatype arrayRefVar[];
```

例如：

```
double myList[];
double[][] a;   //等同于double [][] a;
```

数组变量是引用类型的变量，声明数组引用变量并不分配数组内存空间。必须通过new实例化数组来分配数组内存空间。

创建数组-new

使用new操作符创建数组：

```
arrayRefVar = new datatype[arraySize];
```

例如：

```
myList = new double[10]; //这时才分配内存
```

声明和创建在一条语句中：

```
datatype[] arrayRefVar = new datatype[arraySize];
```

或者

```
datatype arrayRefVar[] = new datatype[arraySize];
```

例如：

```
double[] myList = new double[10];
```

或者

```
double myList[] = new double[10];
```

数组元素初始化

新创建的数组对象，其元素根据类型被设置为默认的初始值（实际上都为0）。

- 数值类型为`0`
- 字符类型为`'\u0000'`（u后面为十六进制，必须4位写满）
- 布尔类型为`false`
- 引用类型为`null`
  数组可以在声明后的花括号中提供初始值：

```
double[] myList = {1.9, 2.9, 3, 3.5};//可以将int转化为double类型，这时不用指定维度size
```

或者

```
double[] myList;
myList = new double[] {1.9, 2, 3.4, 3.5};
//可以将int转化为double类型，声明和创建不在一条语句时，不能直接用{}来初始化
```

访问数组

数组的大小在创建这个数组之后不能被改变。用以下语法访问数组的长度：

```
arrayRefVar.length
```

例如：`myList.length`的值为10。
数组元素通过索引进行访问。元素的索引从0开始，范围从0到`length - 1`：

```
arrayRefVar[index]
```

例如：`myList[0]`表示数组的第一个元素，`myList[9]`表示数组的最后一个元素。

数组示例

编写程序，读入6个整数，找出它们中的最大值。

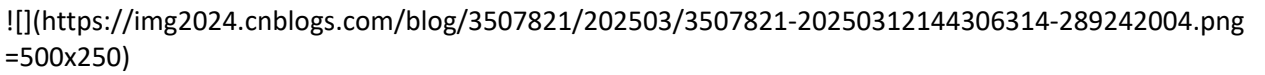
```
public class TestArray {
    public static void main(String[] args) {/** Main method */
        final int TOTAL_NUMBERS = 6;
        int[] numbers = new int[TOTAL_NUMBERS];

        // Read all numbers
```

```
        for (int i = 0; i < numbers.length; i++) {
            String numString = JOptionPane.showInputDialog("Enter a number:");
            numbers[i] = Integer.parseInt(numString);
        }
        // Find the largest
        int max = numbers[0];
        for (int i = 1; i < numbers.length; i++) {
            if (max < numbers[i])   max = numbers[i];
        }
        System.out.println("Max number is " + max);
    }
}
```

7.2 数组的复制

直接使用赋值语句不能实现数组复制，结果是两个数组引用变量指向同一个数组对象（浅拷贝赋值）。



复制数组的方法

使用循环来复制每个元素：

```
int[] sourceArray = {2,3,1,5,10};
int[] targetArray = new int[sourceArray.length];
for(int i = 0; i < sourceArray.length; i++){
    targetArray[i] = sourceArray[i];
}
```

使用`System.arraycopy`方法：`sourceArray`，`targetArray`都已经实例化好。

```
arraycopy(sourceArray,srcPos,targetArray,tarPos,length);
System.arraycopy(sourceArray,0,targetArray,0, sourceArray.length);
```

使用数组的`clone`方法：`targetArray`可先不实例化。

```
int[] targetArray = sourceArray.clone();
```

7.3 将数组传递给方法（数组作为方法参数）

可以将数组变量作为实参传递给方法。基本数据类型传递的是实际值的拷贝，修改形参，不影响实参。数组引用变量传递的是对象的引用，修改形参引用的数组，将改变实参引用的数组。也可以从方法中返回数组，返回的也是引用。

数组传递给方法示例

```
public class TestPassArraySimple{
    /** Main method */
    public static void main(String[] args) {
        int x =1;
        int[] y = new int[10];
        y[0] = 20;

        m(x, y);
```

```
        System.out.println("x is " + x);
        System.out.println("y[0] is " + y[0]);
    }
    public static void m(int number, int[] numbers) {
        number = 1001;       //不改变x的值：值参传递
        numbers[0] = 5001; //改变y[0]
    }
}
```

String、Integer这样的对象作为参数传递要注意的问题

```
public class CallByReferenceException {
    public static void main(String[] args) {
        Integer x = new Integer(10);
        testInteger(x);
        System.out.println("x = " + x);

        String y = "ABC";
        testString(y);
        System.out.println("y = " + y);
    }
    public static void testInteger(Integer i) {
        i = 20;
        System.out.println("i = " + i);
    }
    public static void testString(String s) {
        s = "abc";
        System.out.println("s = " + s);
    }
}
```

输出结果：

```
i = 20
x = 10
s = abc
y = ABC
```

原因是`String`、`Integer`的内容是不可更改的。在`Integer`内部，用`private final int value`来保存整数值；在`String`内部，用`private final char value[]`来保存字符串内容。对于`String`、`Integer`这样内容不可改变的对象，当对其赋值时实际上创建了一个新的对象。可以通过debug来观察对象引用。

7.4 从方法中返回数组

调用方法时，可向方法传递数组引用，也可从方法中返回数组引用。下面的方法返回一个与输入数组顺序相反的数组引用：

```
public static int[] reverse (int[] list){
    int[] result = new int [ list.length ];
    for(int i = 0, j = result.length - 1; i < list.length; i++,j--){
        result [ j ] = list [i];
    }
    return result;
}
```

```
int[] list1 = {1, 2, 3, 4, 5, 6};
int[] list2 = reverse(list1);
```

7.5 可变长参数列表

可以把类型相同但个数可变的参数传递给方法。方法中的可变长参数声明如下：

```
typeName … parameterName
```

在方法声明中，指定类型后面跟省略号。只能给方法指定一个可变长参数，同时该参数必须是最后一个参数。Java将可变长参数当数组看待，通过`length`属性得到可变参数的个数。

```
print(String... args){   //可看作String []args
    for(String temp:args)
        System.out.println(temp);
    System.out.println(args.length);
}
```

调用该方法：

```
print("hello","lisy");
```

7.6 数组的查找和排序

数组的查找-线性搜索法

线性搜索法(linear searching)将一个值与数组的每个元素进行比较。如果找到相同的元素，返回元素的索引；否则返回-1。最坏情况下需要比较N次，平均要比较N/2次，效率不高，时间复杂度O(N)。

```
/** The method for finding a key in the list */
public static int linearSearch(int[] list, int key) {
    for (int i = 0; i < list.length; i++)
        if (key == list[i])                return i;
    return -1;
}
```

数组的查找-二分搜索法

二分搜索法(binary searching)是在一个已排序的数组中搜索特定元素。假设数组已按升序排列，将关键字与数组中间元素进行比较：

- 如果关键字比中间元素小，则在前一半数组中搜索；
- 如果关键字与中间元素相同，查找结束；
- 如果关键字比中间元素大，则在后一半数组中搜索。
  二分法每比较一次就排除一半元素。假设数组有N个元素，为讨论方便，设N是2的幂指数。经过第1次比较，剩下N/2个元素需要查找，经过第2次，剩下N/2/2个元素。经过k次，剩下N/2k个元素。当$k=\log_2 N$时，只剩下一个元素。所以最坏情况下该算法需要比较$\log_2 N + 1$次。假设N = 1024（2^{10}），最多只需要比较11次，而线性查找最坏需要1024次。因此算法的复杂度$O(\log_2 N)$。

```
/** Use binary search to find the key in the list */
public static int binarySearch(int[] list, int key) {
    int low = 0;
    int high = list.length - 1;

    while (high >= low) {
        int mid = (low + high) / 2;
        if (key < list[mid])
            high = mid - 1;
        else if (key == list[mid])
            return mid;
        else
            low = mid + 1;
    }
    return  - 1;
}
```

数组的排序-选择排序算法

假设要将数组按升序排列：

- 将列表中的元素最大值放在最后一个位置；
- 将剩下元素的最大值放在倒数第二的位置；
- 以此类推，直到剩下一个数为止。

```
static void selectionSort(double[] list) {
    for (int i = list.length - 1; i >= 0; i--) {
        // Find the maximum in the list[0..i]
        double currentMax = list[0];
        int currentMaxIndex = 0;

        for (int j = 0; j <= i; j++) {
            if (currentMax < list[j]) {
                currentMax = list[j];
                currentMaxIndex = j;
            }
        }

        // Swap list[i] with list[currentMaxIndex] if necessary;
        if (currentMaxIndex != i) {
            list[currentMaxIndex] = list[i];
            list[i] = currentMax;
        }
    }
}
```

7.7 Arrays类

`java.util.Arrays`类包括各种静态方法，其中实现了数组的排序和查找。

- 排序：

```
double[] numbers={6.0, 4.4, 1.9, 2.9};
java.util.Arrays.sort(numbers); //注意直接在原数组排序
```

- 二分查找：

```
int[] list={2, 4, 7, 10, 11, 45, 50};
int index = java.util.Arrays.binarySearch(list, 11);
```

Arrays和String是常用的两个值得研究的类。

7.8 命令行参数

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，用双引号括起来。格式：

```
java 类名 参数1 参数2 …
```

例如：java TestMain “First number” alpha 53
命令行参数将传递给main方法的args参数。args是一个字符串数组，可以通过数组下标访问每个参数。

```
public static void main(String[] args)
```

注意Java的命令行参数不包括类名，args.length==3。可变长参数用…定义。args是一个字符串数组，可以定义为可变长参数。String … args可以当成String[] args数组使用。

```
public static void main(String … args)  //也可以作为启动函数
```

注意在定义重载函数时，编译器认为String[] args和String … args类型相同。

7.9 多维数组

二维数组：数组的数组

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)
```

在一条语句中声明和创建数组：

```
dataType[][] refVar = new dataType[rowSize][colSize];
```

或者

```
dataType refVar[][] = new dataType[rowSize][colSize];
```

二维数组的长度

二维数组的每个元素是一个一维数组。例如int[][] X =new int[3][4];：

- X指向的是内存里的一个一维数组，数组X的长度是数组X的元素的个数，可由X.length得到，X.length = 3。
- 元素X[i]是引用，指向另一个一维数组，其长度可由X[i].length得到。

- `X.length`是`final`的，不可修改。

不规则数组

二维数组每一行的的列数可以不同。创建不规则二维数组时，可以只指定第一维下标。这时第一维的每个元素为`null`，必须为每个元素创建数组。例如：

```
int[][] x = new int[5][ ];  //第一维的每个元素为null
x[0] = new int[5];//为每个元素创建数组
x[1] = new int[4];
x[2] = new int[3];
x[3] = new int[2];
x[4] = new int[1];
//x.length=5
//x[2].length=3
//x[4].length=1 ，只能取x[4].length的值(它是final的）
```

在C++里如何创建不规则数组

```
//创建二维不规则的动态数组
const int length = 10;
int ** a = (int **)malloc(length * sizeof(int *));
for (int i = 0; i < length; i++) {
    a[i] = (int *)malloc((length - i) * sizeof(int));
    memset(a[i], 0, (length - i) * sizeof(int));
//malloc出来的内存，值是随机的，因此用memset把内存全部设为0
}

for (int i = 0; i < length; i++) {
    for (int j = 0; j < length - i; j++) {
        printf("%2d ", a[i][j]);
    }
    printf("\n");
}
//别忘了释放内存是C++程序员的责任，这是个很痛苦的事
for (int i = 0; i < length; i++) {
    free(a[i]); a[i] = 0;
}
free(a); a = 0;
```

第9章 对象和类

9.1 类和对象的UML表示

C面向过程(或函数)设计，而Java面向对象设计。**对象(object)**是现实世界中可识别(不一定可见)的实体，对象具有状态和行为。其状态是其属性的当前值，其行为是一系列方法，这些方法可改变对象的状态。对象示例：学生、按钮、政府等。

类(class)定义或封装同类对象共有的属性和方法，即将同类型对象共有的属性和行为抽象出来形成类的定义。例如开发学生管理系统，所有学生的共有属性和行为如下：

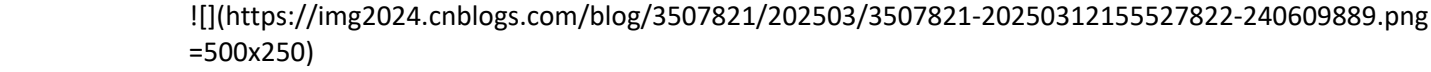
- 属性：学号、姓名、性别、所在学院、年级、班级
- 行为：考试、上课、完成作业

由此形成类的定义：`Class Student{ … }`，属性作为数据成员，行为作为方法成员。同一类型的对象有相同的属性和方法，但每个对象的属性值不同。类(类型简称)是对象的模板、蓝图，对象是类的实例。当定义好类`Student`，可以用类型`Student`去实例化不同对象代表不同学生，如`Student s = new Student(…)`。

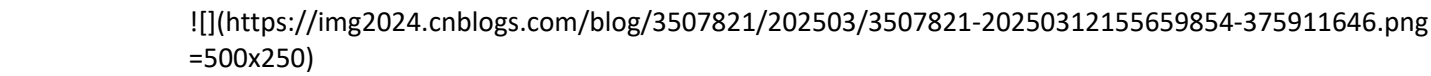
UML是广泛应用的面向对象设计的建模工具，独立于任何具体程序设计语言。作为一种建模语言，UML有严格的语法和语义规范。对于复杂系统，先用UML建模，再编写代码。UML工具会自动把模型编译成Java(C++)源码（方法体是空的）。

UML采用一组图形符号来描述软件模型，这些图形符号简单、直观和规范。所描述的软件模型，可以直观地理解和阅读，由于具有规范性，所以能够保证模型的准确、一致。

- **类的UML表示**：类名在上，数据字段在中间，构造函数和方法在下面。成员访问权限：公有`public`用`+`表示，保护`protected`用`#`表示，私有`private`用`-`表示，包级用`~`表示或默认无表示。包级即可以被同一个`package`的代码访问的成员。Java无`friend`，无析构函数，垃圾自动回收。
- **成员访问权限：**公有public用+表示，保护protected用#表示，
  私有private用-表示，包级用~表示或默认无表示
  包级即可以被同一个package的代码访问的成员。Java无friend，无析构函数，垃圾自动回收



- **对象的UML表示**：对象名:类名，下面是数据字段值。



9.2 定义类并用new创建其对象

圆类及其3个对象：数据字段即圆类属性。Java无`struct`和`union`。

```
class Circle {
    double radius = 1.0;

    Circle() {
        radius = 1.0;
    }

    Circle(double r) {
        radius = r;
    }

    double findArea() {
        return radius * radius * 3.14159;
    }
```

```
}
Circle c1=new Circle(), c2=new Circle(10.0);
```

new会自动调用构造函数，根据实参确定调用哪个构造函数 。

与基本数据类型一样，可声明并用new创建对象数组。如int[]a=new int[10];，所有元素缺省初值=0；当创建对象数组时，数组元素的缺省初值为null。

```
Circle[] circleArray = new Circle[10];
for(int i = 0; i < circleArray.length; i++) {
    circleArray[i] = new Circle( );
}
```


9.3 构造函数(constructor)

- 无返回类型，名字同类名，用于初始化对象。注意JAVA如果定义void className(…)，被认为是普通方法，只在new时被自动执行。
- 必须是实例方法（无static），可为公有、保护、私有和包级权限。
- 类的变量为引用(相当于C指针)，指向实例化好的对象。如Circle c2=new Circle(5.0);，调用时必须有括弧，可带参初始化。
- **缺省构造函数**：如果类未定义任何构造函数，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。
- Java没有析构函数，但垃圾自动回收之前会自动调用finalize( )，可以覆盖定义该函数（但是finalize调用时机程序员无法控制）。

```
public class ConstructorTest {
    public ConstructorTest() {
        System.out.println("constructor");
    }

    public void ConstructorTest() {
        System.out.println("normal instance method return void");
    }
    public double ConstructorTest(double d) {
        System.out.println("normal method return double");
        return d;
    }
    public static void main(String... args){
        new ConstructorTest().ConstructorTest();
    }
}
```

9.4 理解对象访问、向方法传递对象引用

- **访问对象**：通过对象引用访问。JVM维护每个对象的引用计数器，只要引用计数器为0，该对象会由JVM自动回收。通过对象引用，可以：
    - 访问对象的实例变量(非静态数据字段)，如c2.radius。

- 调用对象的实例方法，如`c2.findArea( )`。通过`c2`调用实例方法时，`c2`引用会传给实例方法里的`this`引用。
- 也可访问静态成员和静态方法（不推荐，推荐用类名）。
- 在实例方法中有个`this`引用，代表当前对象(引用当前对象：相当于指针)，因此在实例方法里，可以用`this`引用访问当前对象成员，如`this.radius`、`this.findArea( )`；在构造函数中调用构造函数，须防止递归调用，不能对`this`进行赋值。
- 匿名对象也可访问实例(或静态)成员，如`new Circle( ).radius=2;`

```
public class TestSimpleCircle {
    public static void main(String[] args){
        Circle c1 = new Circle();
        System.out.println("Area = " + c1.findArea() + ", radius = " + c1.radius);

        Circle c2 = new Circle(10.0);
        System.out.println("Area = " + c2.findArea() + ", radius = " + c2.radius);

        c2.setRadius(20.0);
        System.out.println("Area = " + c2.findArea() + ", radius = " + c2.radius);
    }
}

public class Circle {
    double radius = 1.0;

    Circle() {
        radius = 1.0;
    }

    Circle(double r) {
        this.radius = r;
    }

    double findArea() {
        return radius * radius * Math.PI;
    }

    public void setRadius(double newRadius){
        this.radius = newRadius;
    }
}
```

- **引用类型与基本类型区别**：引用变量表示数据的内存单元地址或存储位置，基本类型变量存储的是基本类型的值。数组和类是引用类型变量，它引用了内存里的数组或对象。每个对象（数组）有引用计数，引用类型变量存储的是对象的引用。当变量未引用任何对象或未实例化时，它的值为`null`，一个对象的引用计数`=0`时被自动回收。
- **对象作为方法参数**：对象作为方法参数时与传递数组一样，传递对象实际是传递对象的引用。基本数据类型传递的是实际值的拷贝，传值后形参和实参不再相关：修改形参的值，不影响实参；引用类型变量传递的是对象的引用，通过形参修改对象`object`，将改变实参引用的对象`object`

- Java无类似C++的`&`或C#的`ref`来修饰方法参数，只能靠形参的声明类型来区分是传值还是传引用，因此一定要注意区分。
- **包（package）：**
  - 包是一组相关的类和接口的集合。将类和接口分装在不同的包中，可以避免重名类的冲突，更有效地管理众多的类和接口。`package`就是C++里的`namespace`。
  - 包的定义通过关键字`package`来实现，`package`语句的一般形式：`package 包名;`。`package`语句必须出现在`.java`文件第一行，前面不能有注释行也不能有空白行，该`.java`文件里定义的所有内容（类、接口、枚举）都属于`package`所定义的包里。如果`.java`文件第一行没有`package`语句，则该文件定义的所有内容位于`default`包（缺省名字空间），但不推荐。
  - 不同`.java`文件里的内容都可以属于同一个包，只要它们第一条`package`语句的包名相同。
  - 在同一个`package`里不能定义同名的标识符（类名，接口名，枚举名）。如果要使用其它包里标识符，有二个办法：
    - 用完全限定名，如调用`java.util`包里的`Arrays`类的`sort`方法：`java.util.Arrays.sort(list)`。
    - 在`package`语句后面，先引入要使用其它包里的标识符，再使用。如`import java.util.Arrays;` 或 `import java.util.*;`，`import`语句可以有多条，分别引入多个包里的名字。
  - **使用二种import的区别：**
    - 单类型导入(single type import)：导入包里一个具体的标识符，如`import java.util.Arrays;`，把导入的标识符引入到当前`.java`文件，因此当前文件里不能定义同名的标识符，类似C++里`using nm::id;` 把名字空间`nm`的名字`id`引入到当前代码处。
    - 按需类型导入(type import on demand)：并非导入一个包里的所有类，只是按需导入，如`import java.util.*;`，不是把包里的标识符都引入到当前`.java`文件，只是使包里名字都可见，使得我们要使用引入包里的名字时可以不用使用完全限定名，因此在当前`.java`文件里可以定义与引入包里同名的标识符，但二义性只有当名字被使用时才被检测到，类似于C++里的`using nm;`。

- 包除了起到名字空间的作用外，还有个很重要的作用：提供了`package`一级的访问权限控制（在Java里，成员访问控制权限除了公有、保护、私有，还多了包一级的访问控制；类的访问控制除了`public`外，也多了包一级的访问控制）。
- 包的命名习惯：将Internet域名作为包名（但级别顺序相反），这样的好处是避免包名的重复，如`org.apache.tools.zip`、`cn.edu.hust.cs.javacourse.ch1`。注意包名和实际工程目录之间的对应关系。
- **数据成员的封装**：面向对象的封装性要求最好把实例成员变量设为私有的或保护的，同时为私有、保护的实例成员变量提供公有的`get`和`set`方法。`get`和`set`方法遵循JavaBean的命名规范。设成员为`DateType propertyName`，`get`用于获取成员值：`public DateType getPropertyName( );`；`set`用于设置成员值：`public void setPropertyName(DateType value)`。

```
class Circle{
    private double radius=1.0;
    public Circle( ){ radius=1.0; }
    public double getRadius( ){ return radius; }
    public void setRadius(double r){ radius=r; }
}
```

9.5 实例(或静态)的变量、常量和方法

```
class Circle {
    private double radius;
    private static int numberOfObjects = 0;

    public Circle() { radius = 1.0; numberOfObjects++; }
    public Circle(double newRadius) { radius = newRadius; numberOfObjects++; }

    public double getRadius() {return radius;}
    public void setRadius(double newRadius) { radius = newRadius;}
    public static int getNumberOfObjects() {return numberOfObjects;}

    public double findArea() {  return radius * radius * Math.PI; }
    @Override
    public void finalize() throws Throwable {
        numberOfObjects--;
        super.finalize();
    }
}
```

在每个重载的构造函数里计数器`+1`。覆盖从`Object`继承的`finalize`方法，该方法在对象被回收时调用，方法里对象计数器`-1`。注意该方法调用时机不可控制。`@Override`是注解(annotation)，告诉编译器这里是覆盖父类的方法。使用`@Override`注解有如下好处：

1. 可以当注释用，方便阅读。
2. 编译器可以给你验证`@Override`下面的方法名是否是父类中所有的，如果没有则报错。

- **实例变量(instance variable)**：未用`static`修饰的成员变量，属于类的具体实例(对象)，只能通过对象访问，如“对象名.变量名”。
- **静态变量(static variable)**：用`static`修饰的变量，被类的所有实例(对象)共享，也称类变量。可以通过对象或类名访问，提倡“类名.变量名”访问。

- **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。如`Math`类中的静态常量`PI`定义为：`public static final double PI = 3.14159265358979323846;`。所有常量可按需指定访问权限，不能用等号赋值修改，由于它们不能被修改，故通常定义为`public`。
- **final修饰方法**：
  - `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（见继承）。
  - `final`修饰静态方法时，表示该方法不能被子类隐藏(Hiding)。非`final`静态方法可以被子类隐藏。
  - 构造函数不能为`final`的。
- **方法重载（Overload）、方法覆盖(Override)、方法隐藏(Hiding)**：
  - **方法重载**：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）。
  - **方法覆盖和方法隐藏**：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）。方法覆盖针对实例方法，方法隐藏针对静态方法。

```
public class A {
    public void m(int x, int y) {}
    public void m(double x, double y) {}

    //下面语句报错m(int,int)已经定义，重载函数不能通过返回类型区分
    //    public int m(int x, int y) { return 0;};
}

class B extends A{
    public void m(float x, float y) { }
    public void m(int x, int y) {}

    //注意下面这个语句报错，既不是覆盖（与父类的void m(int,int)返回类型不一样）
    // 也不是合法的重载（和父类的m(int,int)参数完全一样，只是返回类型不一致
```

```
    //    public int m(int x, int y) {} //错误

    //子类定义了新的重载函数int m()
    public int m(){return 0;};
}

class A{
    public void m1(){ }
    public final void m2() { }

    public static void m3() { }
    public final static void m4() { }
}

class B extends A{
    //覆盖父类A的void  m1()
    public void m1(){ }

    //下面语句报错，不能覆盖父类的final 方法
    //    public void m2(){ }

    public static void m3() { }
    //下面语句报错，父类final 静态方法不能被子类隐藏
    //    public static void m4() { }
}

B o = new B();
//如果通过对象去访问静态方法m3,永远只能调用B类的m3，将A类的m3隐藏了
o.m3();
```

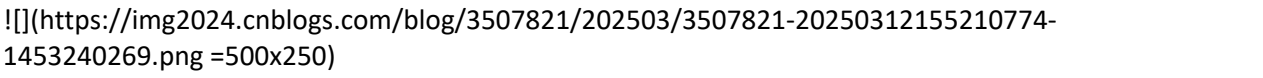
- **静态方法(static method)**：用`static`修饰的方法。构造函数不能用`static`修饰，静态函数无`this`引用。每个程序必须有`public static void main(String[])`方法。静态方法可以通过对象或类名调用，静态方法内部只能访问类的静态成员
  (因为实例成员必须有实例才存在，当通过类名调用静态方法时，可能该类还没有一个实例），静态方法没有多态性。

```
class A{
    int i = 0;
    static int j = 0;
    public static void f(){
        j = 20;
        // i = 20 ; //错误，静态方法不能访问实例变量，调用实例方法
    }
}
```

9.6 可见性修饰符

- **类访问控制符与成员访问控制符**：类访问控制符有public和包级(默认)；类的成员访问控制符有private、protected、public和包级(默认)。Java继承时无继承控制（都是公有继承，和C++不同），父类成员继承到派生类时访问权限保持不变（除了私有）。

- **成员访问控制符的作用**



- **示例代码分析**

```
package p1;
public class C1{//在C1.java
    public int x=1;
    int y=2;//包级
    protected int u=3,w=4;
    private int z;
    public void m1(){
        int i = x = u;
        int j = y = w;
        int k = z;
        m2();
        m3();
    }
    void m2(){ } //包级
    private void m3(){ }
}
```

在上述代码中，`C1`类定义了不同访问权限的成员变量和方法。在`m1`方法中，可访问各类成员。

```
public class C2 extends C1{
//在C2.java
    int u=5; //包级
    void aMethod(){
        C1 o = new C1( );//ok,C1是public
        int i = o.x;//ok, x是public
        int j = o.y;//ok，y(包级)，可在同一包内访问
        int h = o.u;//ok，u(保护)可在同一包内访问
        i=u+super.u;//ok，本类u及super.u(父类保护）
        int k = o.z;//error，z是私有的
        o.m1(); //ok, m1是public
        o.m2(); //ok, m2无访问修饰，可在同一包内访问
        o.m3(); //error, m3是私有的
    }
}
```

`C2`类继承自`C1`，在`aMethod`方法中，可访问`C1`类中部分成员，受访问权限限制。

```
package p2;
public class C3 extends C1{ //C3.java
    int u=5;
    void aMethod(){
        C1 o =new C1( );//ok,C1是public
        int i = o.x; //ok, x 是public
        int j = o.y;//error，y(包级)，不能在不同包内访问
        int h = o.u;//error，u(保护，当前对象非o子类对象)，不能在不同包内访问
        i=u+super.u;//ok，本类u及super.u(保护，当前对象是super的子类对象）
        int k = o.z;//error，z是私有的
```

```
        o.m1(); //ok, m1是public
        o.m2(); //error, m2(包级)，不能在不同包内访问
        o.m3(); //error, m3是私有的
    }
}
```

在不同包的C3类中，访问权限限制更为明显，无法访问C1类中包级和特定保护成员。

- **特殊情况说明**：子类类体中可以访问从父类继承来的protected成员。但如果子类和父类不在同一个包里，子类里不能访问另外父类实例（非继承）的protected成员。

- **类访问控制示例**

```
package p1;
public class C2{
    //可访问同一包的C1类
    C1 c; //OK
}
//C1无访问修饰符，只能在同一包内被访问
class C1{
   …
}
```

```
package p2;
public class C3{
    //不可访问包package p1中的C1类
    C1 c; //error
    //可访问包package p1中的C2类（public）
    C2 c; //OK
}
```

展示了不同包中类的访问权限情况，无修饰符的类只能在同一包内被访问，public类可被其他包访问。

- **访问控制针对类型而非对象级别**

```
public class Foo{
    private boolean x;

    public void m(){
        Foo foo = new Foo();
        //因为对象foo在Foo类内使用，所以可以访问foo的私有成员x，并不是只能访问this.x
        boolean b = foo.x //ok
    }
}
public class Test{
    public static void main(String[] args){
        Foo foo = new Foo();
        //因为对象foo在Foo类外使用，所以不可以访问foo的私有成员x
        boolean b = foo.x //error
    }
}
```

说明在类内部可访问同类对象的私有成员，在类外部则受访问权限限制。

- **构造函数的访问权限**：大多数情况下，构造函数应该是公有的。有些特殊场合，可能会防止用户创建类的实例，这可以通过将构造函数声明为私有的来实现。例如，包java.lang中的Math类的构造函数为私有的，所有的数据域和方法都是静态的，可以通过类名直接访问而不能实例化Math对象。

```
private Math（）{   }
```

9.7 类成员变量的作用域和访问优先级

- **作用域**：类的成员变量(实例变量和静态变量)的作用域是整个类，与声明的位置无关。
- **初始化依赖**：如果一个成员变量的初始化依赖于另一个变量，则另一个变量必须在前面声明。

```
public class Foo {
    int i;//成员变量默认初始化，new后成员默认值为0或null，函数局部变量须初始化
    int j = i + 1;
    int f( ){ int i=0; return i+this.i; } //局部变量i会优先访问
}
```

在上述代码中，`j`的初始化依赖于`i`，虽然`i`在`j`之前声明，但由于成员变量默认初始化，所以不会报错。在`f`方法中，局部变量`i`优先访问，若要访问成员变量`i`，需使用`this.i`。

- **同名变量访问优先级**：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(可用this.i或类名.i发现)。嵌套作用域不能定义同名的局部变量；但类的成员变量可以和类的方法里的局部变量同名。

9.8 this引用

- **指向当前对象**：this引用指向调用某个方法的当前对象。
- **访问实例变量**：在实例方法中，实例变量被同名局部变量或方法形参隐藏，可以通过this.instanceVariable访问实例变量。

```
class Foo{
    int i = 5;
    static double k = 0.0;
    void setI(int i){
        this.i= i;
    }
    static void setK(int k){
        Foo.k = k;
    }
}
```

在`setI`方法中，`this.i`表示类的成员变量`i`，通过`this`可区分局部变量和成员变量。

- **调用构造函数**：调用当前类的其它构造函数，需防止递归调用。使用this(actualParameterListopt)，且必须是构造函数的第1条语句。

```
class Bar{
    int num;
    Bar(){
        this(1); //调用有参构造函数
```

```
    }
    Bar(int n){
        num = n;
    }
}
```

在无参构造函数中，通过`this(1)`调用有参构造函数进行初始化，且`this(1)`必须放在第一行。

以下是对笔记内容的整理和改善，使其格式更清晰，更易阅读和理解：

第11章 继承和多态

11.1 类继承、子类和父类的isA关系

11.1.1语法

```
class ClassName extends Superclass {
    // class body
}
```

- 如果父类是`Object`，则`extends`部分可省略。
- 如果`class C1 extends C2`，则称`C1`为子类（subclass），`C2`为父类（superclass）。
- 子类继承了父类中可访问的数据和方法，子类也可添加新的数据和方法。
- 子类不继承父类的构造函数。
- 一个类只能有一个直接父类（Java不支持多重继承）。
- Java的继承都是公有继承，因此父类的成员如果被继承到子类，访问权限不变。
- 子类和父类是`isA`关系：一个子类对象`isA`父类对象。

1. 设计示例：几何对象类

假设要设计模拟几何对象的类，如圆和矩形，考虑的因素有颜色、是否填充、创建日期、圆的半径、矩形的周长等。

设计思路：

- 不要直接定义`Class Circle`和`Class Rectangle`，先分析几何对象的共同属性和行为。
- 共同的属性和行为有颜色、是否填充、创建时间，以及这些属性的`getter`和`setter`行为。
- 圆的半径、矩形的长宽不是几何图形共有的属性。
- 可以设计通用类`GeometricObject`来模拟共有的属性和方法。
- `Circle`、`Rectangle`类通过继承`GeometricObject`获得共同的属性和行为，同时添加自己特有的属性和行为。

注意： 任何类在设计时应考虑覆盖祖先类 `Object` 的如下函数：`equals`、`clone`、`toString` 等。

2. 代码示例

```
public class GeometricObject { // 等价于 public class GeometricObject extends Object
    private String color = "white";
    private boolean filled;
    private Date dateCreated; // java.util.Date 是 JDK 定义的类，表示日期和时间

    public GeometricObject() { dateCreated = new Date(); }

    public String getColor() { return color; }
    public void setColor(String color) { this.color = color; }
    public boolean isFilled() { return filled; }
    public void setFilled(boolean filled) { this.filled = filled; }
    public Date getDateCreated() { return dateCreated; }

    @Override // 覆盖 Object 类的 toString() 方法
    public String toString() { // 还应考虑 equals, clone
        return "created on " + dateCreated + "\n\tcolor: " + color + " and filled: "
+ filled;
    }
}

public class Circle extends GeometricObject {
    private double radius; // 新增属性

    public Circle() { }
    public Circle(double radius) { this.radius = radius; }

    public double getRadius() { return radius; }
    public void setRadius(double radius) { this.radius = radius; }

    public double getArea() { return radius * radius * Math.PI; }
    public double getDiameter() { return 2 * radius; }
    public double getPerimeter() { return 2 * radius * Math.PI; }

    // 还应考虑 equals, clone, toString 等函数
}

public class Rectangle extends GeometricObject {
    private double width;
    private double height;

    public Rectangle() { }
    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    public double getWidth() { return width; }
    public void setWidth(double width) { this.width = width; }
    public double getHeight() { return height; }
    public void setHeight(double height) { this.height = height; }
```

```
    public double getArea() { return width * height; }
    public double getPerimeter() { return 2 * (width + height); }

    // 还应考虑 equals, clone, toString 等函数
}
```

11.1.2 类属性和方法总结

GeometricObject 类的属性和方法：

- 私有属性：`color`, `filled`, `dateCreated`
- 公有方法：`getColor`, `setColor`, `isFilled`, `setFilled`, `getDateCreated`

Circle 类的属性和方法：

- 继承自 `GeometricObject` 的公有方法：`getColor`, `setColor`, `isFilled`, `setFilled`, `getDateCreated`
- 自己实现的公有方法：`getRadius`, `setRadius`, `getArea`, `getDiameter`, `getPerimeter`
- 自己的私有属性：`radius`

Rectangle 类的属性和方法：

- 继承自 `GeometricObject` 的公有方法：`getColor`, `setColor`, `isFilled`, `setFilled`, `getDateCreated`
- 自己实现的公有方法：`getWidth`, `setWidth`, `getHeight`, `setHeight`, `getArea`, `getPerimeter`
- 自己的私有属性：`width`, `height`

11.1.3 实例初始化模块

实例初始化块（Instance Initialization Block, IIB）是一个用大括号括住的语句块，直接嵌套于类体中，不在方法内。它的作用就像把它放在了类中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。

作用：

- 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，模块按照在类中出现的顺序执行。
- 初始化模块可以简化构造方法的代码。

代码示例：

```
public class Book {
    private static int numOfObjects;
    private String title;
    private int id;
```

```
    public Book(String title) { this.title = title; }
    public Book(int id) { this.id = id; }

    { numOfObjects++; } // 实例初始化块
}
```

等价于：

```
public class Book {
    private static int numOfObjects;
    private String title;
    private int id;

    public Book(String title) {
        numOfObjects++;
        this.title = title;
    }
    public Book(int id) {
        numOfObjects++;
        this.id = id;
    }
}
```

其他作用：

- 截获异常。

```
interface ISay {
    public abstract void sayHello();
}

public class InstanceInitializationBlockTest {
    public static void main(String[] args) {
        ISay say = new ISay() { // 这里定义了一个实现了 ISay 接口的匿名类
            // 必须使用实例初始化块的另外一种场景：下面语句会抛出异常，所以编译错误
            // private InputStream s = new FileInputStream(new File("C:\1.txt"));
            private InputStream s;

            { // 实例初始化块
                try {
                    s = new FileInputStream(new File("C:\1.txt"));
                } catch (FileNotFoundException e) {
                    e.printStackTrace();
                }
            }

            @Override
            public void sayHello() {
                System.out.println("Hello");
            }
        };
        say.sayHello();
    }
}
```

- 在匿名类中初始化数据成员。

```
interface ISay {
    public abstract void sayHello();
}

public class InstanceInitializationBlockTest {
    public static void main(String[] args) {
        ISay say = new ISay() { // 这里定义了一个实现了 ISay 接口的匿名类
            // final 类型变量一般情况下必须马上初始化，一种例外是：final
实例变量可以在构造函数里再初始化。
            // 但是匿名类又不可能有构造函数，因此只能利用实例初始化块
            private final int j; // 为了演示实例初始化块的作用，这里特意没有初始化常量 j

            { // 实例初始化块
                j = 0; // 在实例初始化块里初始化 j
            }

            @Override
            public void sayHello() {
                System.out.println("Hello");
            }
        };
        say.sayHello();
    }
}
```

- 实例初始化模块只有在创建类的实例时才会调用。
- 定义并初始化类的实例变量等价于实例初始化块：private int id = 0；
- 一个类可以有多个实例初始化块，对象被实例化时，模块按照在类中出现的顺序执行，构造函数最后运行。

```
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
    }
}
```

11.1.4 静态初始化模块

- 静态初始化模块是由static修饰的初始化模块{}，只能访问类的静态成员，并且在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）。
- 在类体里直接定义静态变量相当于静态初始化块

代码示例：

```
public class A{
   //类的属性和方法定义
   {
     //实例初始化模块
   }
   static {
     //静态初始化模块
   }
   public static int i = 0;//直接定义静态变量相当于静态初始化块
}
```

- 一个类可以有多个静态初始化块，类被加载时，这些模块按照在类中出现的顺序执行

```
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
  }
}
```

11.1.5 初始化模块执行顺序:

1. **第一次使用类时装入类**
2. 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。
3. 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。
4. **实例化类的对象时**，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。
5. 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。

6. 执行构造函数函数体。
7. 如果声明类的实例变量时具有初始值，如
   `double radius = 5.0;`
   变量的初始化就像在实例初始化模块中一样，即等价于
   `double radius; { radius = 5.0; }`
8. 如果声明类的静态变量时具有初始值，如
   `static int numOfObjects = 0;`
   变量的初始化就像在静态初始化模块中一样，即等价于
   `static int numOfObjects; static{ numOfObjects = 0; }`

初始化模块执行顺序示例

```
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) "); }
}
```

输出顺序：

```
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
(8)
```

11.2 super关键字

利用`super`可以**显式调用父类的构造函数**。语法为`super(parameters)`，且必须是子类构造函数中的第一条且仅一条语句。如果子类构造函数中没有显式调用父类的构造函数，编译器会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。

`super`可以用于**访问父类的成员**，包括静态和实例成员。语法为`super.data`（访问父类属性）和`super.method(parameters)`（调用父类方法）。需要注意的是，`super`不能用于静态上下文（即静态方法和静态初始化块中不能使用`super`），且不能使用`super.super.p()`这样的`super`链。

11.2.1 构造方法链

在构造一个类的实例时，会沿着继承链调用所有父类的构造方法，这称为构造方法链。如果子类中没有显式调用父类的构造函数，编译器会自动在子类构造函数第一条语句前加上`super()`。如果一个类自定义了构造函数（不管有无参数），编译器不会自动加上无参构造函数。

11.2.2 无参构造函数

如果一个类没有定义任何构造函数，编译器会自动加上无参构造函数。在为子类添加无参构造函数时，编译器会默认调用父类的无参构造函数`super()`。如果找不到父类的无参构造函数，编译器为子类添加无参构造函数失败，编译报错。因此，如果一个类定义了带参数的构造函数，必须显式定义一个无参构造函数，否则该类将没有无参构造函数。

没有无参构造函数的后果

例如：

```
class Fruit {
    public Fruit(String name) {
        System.out.println("调用Fruit的构造函数");
    }
}

class Apple extends Fruit { }
```

编译器在为`Apple`提供无参构造函数时出错。因为子类`Apple`没有定义任何构造函数，编译器会尝试提供无参构造函数`Apple()`，并调用父类的无参构造函数`Fruit()`。然而，父类`Fruit`定义了有参构造函数，编译器没有为其提供无参构造函数`Fruit()`，因此编译失败。

11.3 实例方法覆盖

11.3.1 方法覆盖的定义

如果子类重新定义了从父类中继承的实例方法，称为方法覆盖（method override）。仅当父类方法在子类里是可访问的，该实例方法才能被子类覆盖。父类私有实例方法不能被子类覆盖，且父类实例私有方法自动视为`final`的。

1. 静态方法的隐藏

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。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）。

2. 覆盖的特性

一旦父类中的实例方法被子类覆盖，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）。因为实例方法具有多态性（晚期绑定）。在子类函数中可以使用`super`调用被覆盖的父类方法。

3. 隐藏的特性

隐藏特性指父类的变量（实例变量、静态变量）和静态方法在子类被重新定义。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）。

4. 方法覆盖的哲学涵义

子对象当然可以修改父类的行为（生物进化除了遗传，还有变异）。

11.3.2 示例代码

```
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 {
    // 覆盖父类实例方法
    public void m() {
        System.out.println("B's m");
    }
    // 隐藏父类静态方法
    public static void s() {
        System.out.println("B's s");
    }
}

public class OverrideDemo {
    public static void main(String[] args) {
        A o = new B();  // 父类型变量引用子类对象
        o.m();  //
由于父类实例方法m被子类覆盖，o运行时指向B类对象，由于多态性，执行的是B的m
        o.s();  //
由于s是静态方法，没有多态性，编译器编译时对象o的声明类型是A，所以执行的是A的s
    }
}
```

1. 多态性与静态方法

父类型变量o引用了子类对象，通过o调用被覆盖的实例方法m时，调用的一定是子类方法，这时不可能调用到父类的方法m（父类函数不能被发现），因为多态特性。多态性使得根据new后面的类型决定调用哪个m。静态方法和成员变量没有多态性，因此要根据声明类型决定调用哪个s。

2. 引用变量的类型

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**）。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。

3. 强制类型转换

一旦引用变量o指向了B类型对象（A o = new B()），o.m()调用的永远是B的m，再也无法通过o调用A的m，哪怕强制转换都不行：((A)o).m();调用的还是B的m。

4. 静态方法隐藏示例

```
public class OverrideDemo {
    public static void main(String[] args) {
        // 静态方法隐藏
        B o = new B();
        o.s(); // 调用B的s，将父类A的s隐藏，即通过B类型的引用变量o是不可能调用A的s
        ((A)o).s(); //
通过强制类型转换，可以调用A的s，可以找回。也可以通过类名调用来找回:A.s();
    }
}
```

覆盖toString()方法示例

```
public String toString() {
    return "A circle " + super.toString() + "\n\tradius: " + radius;
}
```

这样做的好处是Circle对象的基本属性如color、filled、dateCreated由父类方法打印，Circle对象只负责打印新的属性值。

11.4 Object类中的方法

11.4.1 概述

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。它提供了以下方法：

- toString
- equals
- getClass

- `clone`
- `finalize`

其中，前三个方法为公有，后两个为保护。`getClass`方法为`final`，用于泛型和反射机制，禁止覆盖。

1. equals方法

`equals`方法用于测试两个对象是否相等。`Object`类的默认实现是比较两个对象引用是否引用同一个对象。

覆盖`equals`方法时的注意事项：

1. 对于基本类型数值成员，直接使用`==`判断即可。
2. 对于引用类型变量成员，则需要对这些变量成员调用`equals`判断，不能用`==`。
3. 覆盖`equals`方法时，最好同时覆盖`hashCode()`方法。
4. 首先使用`instanceof`检查参数的类型是否和当前对象的类型一样。

示例：

```
public boolean equals(Object o) {
    if (o instanceof Circle)  // 应先检查另一对象o的类型
        return radius == ((Circle) o).radius;
    return false;
}
```

2. toString方法

`toString`方法返回代表这个对象的字符串。`Object`类的默认实现是返回由类名、`@`和`hashCode`组成。

示例：

```
Circle circle = new Circle();
circle.toString();  // 输出: Circle@15037e5
```

通常子类应该覆盖该方法，提供更有意义的信息。

3. clone方法

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`接口是一个标记接口，没有定义任何方法，只是用来标记一个类是否支持克隆。

覆盖`clone`方法的注意事项：

1. 子类覆盖`clone`方法时应该提升为`public`。
2. 方法里应实现深拷贝，`Object`的`clone`实现是浅拷贝（按成员赋值）。

示例：

```
@Override
public Object clone() throws CloneNotSupportedException {
    return super.clone();  // 调用Object的clone
}
```

11.4.2 对象的浅拷贝与深拷贝

浅拷贝：

- 对于基本类型数值成员，使用=赋值即可。
- 对于引用类型成员，浅拷贝会导致两个对象引用同一个对象。

深拷贝：

- 对于引用类型成员，需要进一步嵌套调用该成员的克隆方法进行赋值。

示例：

```
public Object clone() throws CloneNotSupportedException {
    A newObj = (A) super.clone();
    newObj.values = this.values.clone();  // 数组的clone是深拷贝
    return newObj;
}
```

1. 示例代码

```
class A implements Cloneable {
    public static final int SIZE = 10;
    private int[] values = new int[SIZE];

    public int[] getValues() {
        return values;
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        A newObj = (A) super.clone();
        newObj.values = this.values.clone();
        return newObj;
    }

    public boolean equals(Object obj) {
        if (obj instanceof A) {
            A o = (A) obj;
            return java.util.Arrays.equals(this.getValues(), o.getValues());
        } else return false;
    }

    public String toString() {
        StringBuffer buf = new StringBuffer();
        for (int v : values) {
            buf.append(v + " ");
        }
        return buf.toString().trim();
    }
```

```
}

public class CloneTest {
    public static void main(String[] args) throws CloneNotSupportedException {
        A o1 = new A();
        o1.setValues(new int[]{1, 2, 3, 4, 5, 6, 7, 8, 9, 10});
        A o2 = (A) (o1.clone());
        System.out.println(o1 == o2);  // false
        System.out.println(o1.getValues() == o2.getValues());  // false
        System.out.println(o1.equals(o2));  // true
        System.out.println(o2.toString());  // 显示 1 2 3 4 5 6 7 8 9 10
    }
}
```

2. 克隆的深度问题

如果类中包含引用类型成员，只要该成员实现了深拷贝克隆，则外层类可以很方便地实现深拷贝克隆。

示例：

```
class B implements Cloneable {
    A a;  // 引用类型成员
    int i;  // 值类型

    @Override
    public Object clone() throws CloneNotSupportedException {
        B newObj = (B) super.clone();
        newObj.i = this.i;  // 值类型成员直接=赋值
        newObj.a = (A) (this.a.clone());  //
引用类型的成员不能直接赋值，必须调用clone方法
        return newObj;
    }
}
```

通过这种方式，可以实现多层嵌套的深拷贝克隆。

11.5多态性、动态绑定和对象的强制类型转换

11.5.1 多态性

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（沿着继承链从祖先类到后代类，特征越来越具体；反过来，从后代类往祖先类回溯，越来越抽象）
每个子类的实例都是父类的实例（子类对象ISA父类），但反过来不成立

```
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
```

多态：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 GreetingSender{
       public void newYearGreeting (Person p){ p.Greeting();
//编译时应该是Person的Greeing}
}
public class GreetingTest1{
       public static void main(String[] args){
       GreetingSender g = new GreetingSender();
              g.newYearGreeting(new Person());        //调用Person的Greeting()
              g.newYearGreeting(new Employee());      //调用Employee的Greeting()
              g.newYearGreeting(new Manager());       //调用Manager的Greeting()
      }
}
//以最后一条语句为例来解释多态特性：
//当实参new Manager()传给形参Person p时，等价于Person p = new Manager(),
因此执行p.Greeting()语句时根据形参p指
//向的对象的实际类型动态计算Greeting方法的入口地址，调用了Manager的Greeting()
```

仔细观察程序，可以发现产生多态的三个重要因素：
1：不同类之间有继承链
2：newYearGreeting方法的参数类型用的父类类型
3：newYearGreeting调用的Greeting方法都被子类用自己的行为覆盖
满足了这三个条件，用继承链中不同子类的对象做为方法的实参去调用方法会使该方法表现出不同的行为。由于子类的实例也是父类的实例，所以用子类对象作为实参传给方法中的父类型的形参是没有问题的。

这段程序的微妙之处在于：
GreetingSender类的newYearGreeting方法的参数是Person类型，那么
newYearGreeting的行为应该是Person对象的行为。
但是在实际运行时我们看到随着实参对象类型的变化， newYearGreeting
方法却表现出了多种不同的行为，这种机制称为多态

- **多态条件**：父类变量可引用本类和子类对象，子类对象isA父类对象
- 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o是类C1的实例，C1是C2的子类，C2是C3的子类，…，Cn-1是Cn的子类。也就是说，Cn是最一般的类，C1是最具体的类。在Java中，Cn是Object类。如果调用继承链里子类型C1对象o的方法p，Java虚拟机按照C1、C2、…、Cn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)。
  ![image](https://img2024.cnblogs.com/blog/3507821/202505/3507821-20250516155507823-891417063.png =500x250)

查找方法p的顺序：看C1是否覆盖p，如果已覆盖，调用C1的p;如果C1没有覆盖p，则查看C2是否覆盖，以此类推
从C1开始顺着继承链往父类查找，直到找到第一个p的实现，并调用这个p的实现

11.5.2 强制类型转换

1. 概念

父类变量引用子类对象，可视为将子类对象转换为父类（不需强制类型转换）。
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`
从父类到子类必须显式（强制）转换。
`Manager m = p; //编译错 ，p是Person父类型，Person不一定是Manager`
`Manager m = (Manager)p;//ok，但运行时没有检查`
从父类到子类转换必须显式转换，用**instanceof**进行运行时类型检查更安全。
Manager m = null;
if(p instanceof Manager) m= (Manager)p; //安全：进行运行时类型检查

为什么从父类到子类转换必须强制类型转换？
首先要理解**类型检查**（type checking）发生在编译时，然后要理解Person p = new Manager()的真正涵义
![image](https://img2024.cnblogs.com/blog/3507821/202505/3507821-20250516155946091-920941455.png =500x250)

但是创建Manager对象并由p来引用是在运行时发生，因为程序还没运行，编译器无法知道p会指向什么对象，编译器在编译时只能根据变量p的声明类型（Person）来类型检查
`Person p=new Manager();`
当编译器检查到 Manager m = p；编译器认为Person类型引用p要赋值给类型为Manager类型引用，扩展内存可能引起麻烦且不安全，因此，编译器认为类型不匹配，会报错。
加上强制转换 Manager m = (Manager)p；意思是强烈要求编译器，把p解释成Manager类型，风险我来承担。这个时候编译器就按Manager类型来解释p
因此，强制类型转换意味着你自己承担风险，编译器不会再做类型检查。
强制类型转换的风险是：运行时如果p指向的对象不是Manager的实例时程序会出错。
为了避免风险，最好在运行时用instanceof来做实例类型检查。

2. instanceof操作符

可以用instanceof操作符判断一个引用指向的对象是否是一个类的实例。表达式返回boolean值（发生在运行时）。
例如：

```
Person p = new Manager();
if（p instanceof Manager）
      Manager m = （Manager）p
```

意思是如果p指向的对象真的是Manager实例，p可以非常安全的赋值给m
注：所谓的强制类型转换发生在编译时：要求编译器将p的类型由Person解释成Manager，将上面的语句放行，后果程序员自负
当运行时，代码变为`Manager m = p;`

由于加了if判断，保证了运行时p所引用的对象的类型一定是Manager，因此运行时不会有问题。但是如果没有if语句的条件判断，当运行时p所引用的对象类型不是Manager，运行时就会报错

总结

- 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实例方法。
- 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。
- 例子：
  编写程序，创建两个几何对象：圆和矩形。调用displayObject来显示结果。
  如果对象是圆，显示半径和面积
  如果对象是矩形，显示面积
  警告: 对象访问运算符(.)优先于类型转换运算符。使用括号保证在(.)运算符之前转换
  ((Circle)object).getArea() //OK
  (Circle)object.getArea(); //错误

11.6访问控制符和修饰符final

1. `final` 修饰变量

`final` 修饰的变量表示常量，一旦初始化后其值不能被修改。

(1) `final` 成员变量

- **特点**：必须在声明时初始化，或者在构造方法中初始化。
- **示例**：

```
public class Example {
    final int CONSTANT_VALUE = 10; // 声明时初始化
    final int anotherValue;

    public Example(int value) {
        anotherValue = value; // 在构造方法中初始化
    }
}
```

  - 如果尝试修改 `CONSTANT_VALUE` 或 `anotherValue`，编译器会报错。

(2) `final` 局部变量

- **特点**：可以在声明时初始化，也可以在后续代码中初始化（但只能初始化一次）。
- **示例**：

```
public void method() {
    final int localVar;
    localVar = 20; // 初始化
    // localVar = 30; // 错误：不能再次赋值
}
```

---

2. `final` 修饰方法

`final` 修饰的方法表示最终方法，不能被重写（覆盖）或隐藏。

(1) 实例方法

- **特点**：子类不能重写该方法。
- **示例**：

```
class Parent {
    final void display() {
        System.out.println("Parent's display method");
    }
}

class Child extends Parent {
    // 错误：不能重写 final 方法
    // void display() { ... }
}
```

(2) 静态方法

- **特点**：子类不能隐藏该方法。
- **示例**：

```
class Parent {
    static final void show() {
        System.out.println("Parent's show method");
    }
}

class Child extends Parent {
    // 错误：不能隐藏 final 静态方法
    // static void show() { ... }
}
```

---

3. `final` 修饰类

`final` 修饰的类表示最终类，不能被继承。

- **特点**：该类不能有子类。
- **示例**：

```
final class FinalClass {
    void method() {
        System.out.println("Final class method");
    }
}

// 错误：不能继承 final 类
// class SubClass extends FinalClass { ... }
```

常见 `final` 类

- `String`：不可变字符串类。
- `StringBuffer`：线程安全的可变字符串类。
- `Math`：提供数学运算的工具类。

---

4. `Object` 类的 `getClass()` 方法

`getClass()` 是 `Object` 类的一个 `final` 方法，用于获取对象的运行时类。

- **特点**：不能被重写。
- **示例**：

```
class Example {
    public static void main(String[] args) {
        String str = "Hello";
        Class<?> cls = str.getClass();
        System.out.println(cls.getName()); // 输出：java.lang.String
    }
}
```

---

5. 补充说明

- **`final` 与不可变性**：
  - `final` 修饰引用类型变量时，变量的引用不能改变，但对象的内容可以改变。
  - 示例：

```
final StringBuilder sb = new StringBuilder("Hello");
sb.append(" World"); // 合法：修改对象内容
// sb = new StringBuilder(); // 错误：不能修改引用
```

- **`final` 与性能优化**：
  - `final` 变量在编译时可能会被内联优化，提高性能。

总结

修饰目标	特点	示例
变量	初始化后不可修改	`final int x = 10;`
实例方法	不能被子类重写	`final void method() { ... }`
静态方法	不能被子类隐藏	`static final void show() { ... }`
类	不能被继承	`final class FinalClass { ... }`

第12章 异常处理和文本I/O

12.1 异常处理概述

12.1.1 异常定义

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) 。

12.1.2 异常定义异常产生的原因

- 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：
    1. 表达式违反了正常的语义，例如整数除零。
    2. 通过空引用访问实例变量或方法。
    3. 问数组超界。
    4. 资源超出了某些限制，例如使用了过多的内存。
- **显式**地执行throw语句抛出异常

异常的抛出都是由throw语句直接或间接抛出：
1：程序运行时的逻辑错误导致异常间接抛出，例如通过空引用访问实例变量和方法

```
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();
   }
}
```

2：程序在满足某条件时，用throw语句直接抛出异常，如
if(满足某条件){
throw new Exception(“异常描述信息”);

```
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
       }
   }
}
```

> 为什么这里的main函数必须加异常声明而前一个PPT例子不需要？
> 一个是必检异常，一个不是

12.1.3 异常分类

Java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。
Java的异常分为二大类：从Exception派生的是程序级错误，可由程序本身处理；从Error派生是系统级错误，程序可不用处理（也基本上处理不了，例如JVM内存空间不够）。
Exception的子类里，除了**RuntimeException**这个分支外，其他的都是必检异常（即：要么在函数里用catch子句捕获并处理，要么在所在函数加上异常声明，PPT第5页例子）。
RuntimeException的子类是非必检异常（PPT第4页例子）

![image](https://img2024.cnblogs.com/blog/3507821/202505/3507821-20250516235416340-1950401910.png =500x250)

12.1.4 异常处理

运行时异常系统处理异常的过程如下：

- 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

12.2 异常声明、抛出和捕获

12.2.1 异常声明

- **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。

- 其它的异常称为**必检异常**(Checked Exception)，编译器确保必检异常被捕获或声明（即要不在方法里捕获异常，要不在方法头声明异常）
    - 捕获：方法可以通过try/catch语句来捕获异常
    - 声明：方法可以在方法头使用throws子句声明可能抛出异常
- 方法可以抛出的异常
  方法里调用throw语句直接抛出的任何异常
  调用另一个方法时，由被调用方法间接抛出的异常
- **异常声明**：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
  例如

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

12.2.2 异常的抛出和捕获

1. 抛出异常

抛出异常有二种情况

- 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
- 显式直接抛出
  例如

```
        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");
        }
```

2. 捕获异常

语法

```
      try {
            statements
      } catch (ExceptionType1 id1) {
            statements1
      } catch (ExceptionType2 id2) {
            statements2
      } finally {
            statements3
      }
```

当包含catch子句时，finally子句是可选的。
当包含finally子句时，catch子句是可选的。

- 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。

12.2.3 方法异常声明与方法内捕获处理异常的关系（对必检异常）

例1

```
public class ThrowDeclaration1 {
    //由于m1内部处理了所有异常，因此不用加throws声明
    public void m1(){
        try{
            //执行可能抛出异常的语句
        }
        catch(Throwable e){ //由于Throwable是所有异常的父类，因此这里可以捕获所有异常
            //处理异常
        }
    }

    public void m2(){
        m1(); //由于m1没有异常声明，因此m1的调用者不需要try/catch
    }
}
```

例2:
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,方法可以不捕获同时不声明非必检异常（注意只是编译器不检查了，但如果真的有异常该抛出还是会抛出）

- 10/0这种异常
- 方法如果声明或捕获非必检异常也没问题

```
public class ThrowDeclaration2 {
    //m1内部可能抛出的异常没有处理，因此必须加throws声明
    //throws声明就是告诉方法的调用者，调用本方法可能抛出什么异常
    public void m1() throws IOException {
```

```
        //执行可能抛出异常IOException的语句,但没有try/catch

    }

    public void m2(){
        //由于m1有异常声明，因此m2调用m1时有第一个选择：1 用try/catch捕获和处理异常
       //这时m2就不用加throws异常声明
        try {
            m1();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
public void m2() throws IOException {
//由于m1有异常声明，因此m2调用m1时有第2个选择：
//2 也在方法头声明异常，方法体里不捕获异常。
//这时如果有方法m3调用m2, m3也就面临二个选择：声明异常或者在m3里捕获异常
m1();
}
```

12.3 异常的捕获顺序

- 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。因为throw异常打断了正常执行次序
- 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。

实例：

```
import java.lang.System;
import java.lang.ArithmeticException;
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;  }
```

```
      //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

        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,是否捕获了异常，finally和return总会执行，因此最后返回-1
        return  r;
    }
    public  static  void  main(String[ ] args)
{//虽然div没有异常声明，在main里调用div也可用try/catch
        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( ));
        }
    }
}
```

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。
例：

```
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I/O

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.PrinterWriter。
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PrinterWriter ： 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。

例:

```
package  filecopy;
import java.lang.System;
import java.io.File;
import java.io.PrintWriter;
import java.io.IOException;
import java.util.Scanner;
public class Copy {
    public static void main(String[ ] args) {
        if(args.length!=2){
            System.out.println("Usage: Java  Copy  <sourceFile>   <tagetFile>");
            System.exit(1);
        };
        File sF=new File(args[0]); //构造源文件File对象，args[0]:源文件路径
        if(!sF.exists( )){
            System.out.println("Source Filel "+args[0]+ "does not exist!");
            System.exit(2);
        };
        File tF=new File(args[1]);   //构造目标文件File对象， args[1]:目标文件路径
        if(tF.exists( )){
            System.out.println("Target File "+args[0]+ "already exist");
            System.exit(3);
        };
        try{
            Scanner input=new Scanner(sF);    //构造Scanner对象读取源文件
            PrintWriter output=new PrintWriter(tF); //构造PrintWriter对象写目标文件

            while(input.hasNext( )){ //逐行读取源文件，逐行写入目标文件
                String s=input.nextLine(); //从源文件读取下一行
                output.println(s);        //打印这一行到目标文件
            }
            input.close( );
            output.close( );
        }
        catch(IOException ioe){
             System.out.println(ioe.toString( ));
       }
    }
}
```

第13章 抽象类和接口

13.1 抽象类

13.1.1 抽象类概念

- 子类继承父类后，通常会添加新的属性和方法。因此沿着继承链越往下继承的子类其属性和方法越来越具体。相反，越上层的祖先类其实现越抽象，甚至无法给出具体实现。一个长方形图形有面积，但其祖先类GeometricObject的getArea()方法可能没法给出具体实现，这时可以定义成抽象方法。
- Java可定义不含方法体的方法，其方法体由子类根据具体情况实现，这样的方法称为**抽象方法**(abstract method)，包含抽象方法的类必须是**抽象类**(abstract class)。
- 抽象类和抽象方法的声明必须加上abstract关键字。
- 抽象方法的意义：加给子类的一个约束。例如Circle类和Rectangle类计算面积必须使用父类规定的函数签名。这样可以充分利用多态特性使得代码变得更通用

例如

```
abstract class GeometricObject{
    //属性和方法定义

    public abstract double getArea();
    public abstract double getPerimeter();
}
```

> 包含抽象方法的类必须是抽象类
> 抽象类和抽象方法必须用abstract关键字修饰
> **没有包含抽象方法的类也可以定义成抽象类**

抽象方法：使用abstract定义的方法或者接口中定义的方法（接口中定义的方法自动是抽象的，可以省略abstract）。

- 一个类C如果满足下面的任一条件，则该类包含抽象方法且是抽象类：
    1. 类C显式地包含一个抽象方法的声明；
    2. 类C的父类中声明的抽象方法未在类C中实现；
    3. 类C所实现的接口中有的方法在类C里没有实现
    4. **只要类C有一个未实现的方法（自己定义的或继承的），就是抽象类**
- 但是，一个不包含任何抽象方法的类，也可以定义成抽象类
- **抽象类不能被实例化**

13.1.2 抽象类性质和规则

- 只有实例方法可以声明为抽象方法（Java里所有实例方法自动是虚函数，因此Java里没有virtual关键字）。
- 抽象类不能被实例化，即不能用new关键字创建对象（即new
  右边的类型不能是抽象类）。
  但是抽象类可以作为变量声明类型、方法参数类型、方法返回类型
  `GeometricObject o = new Circle();//OK 为什么？`
  因为一个**抽象类型引用变量可以指向具体子类的对象**,变量声明类型、方法参数类型、方法返回类型越抽象越好，尽量用抽象类和接口类型,以方法参数类型为例，方法参数类型越抽象，代码越通用
- 抽象类可以定义构造函数，并可以被子类调用（通过super）。
- 抽象类可以定义变量（实例或静态）、非抽象方法并被子类使用
- **抽象类的父类可以是具体类**：自己引入了抽象方法。例如，具体类Object是所有类的祖先父类。

13.2 接口

13.2.1 概念和用法

接口是**公共静态常量**和**公共抽象实例方法**的集合。接口是能力、规范、协议的反映。
接口不是类：(1)不能定义构造函数；(2)接口之间可以多继承，类可implements多个接口。(3)和抽象类一样，不能new一个接口
语法：[modifier] interface interfaceName {
constant_declaration*
abstract_method_declaration*
}
接口中的**所有数据字段**隐含为public static final
接口体中的**所有方法**隐含为public abstract
从JDK8开始，接口可以定义缺省方法、静态接口方法
从JDK9开始，接口可以定义private和private static方法，越来越像抽象类

- 可以在能够使用任何其他数据类型的地方使用接口。
- 接口类型属于引用类型，接口类型的变量可以是：
  5. 空引用(null)
  6. 引用实现了该接口的类的实例
- 接口需要具体的类去实现。类实现接口的语法

```
[modifier] class className [extends superclass][implements interfaceNameList ] {
       member_declaration*
}
```

- 除非类为abstract,所有接口的成员方法必须被实现
- 一个类只能继承一个父类，但可以实现多个接口，多个接口以“，”分开。

13.2.2 接口的继承

- 接口不是类（Java支持单继承类），**一个接口可以继承多个接口**。
  语法

```
[modifier] interface interfaceName [extends interfaceNameList] {
      declaration*
}
```

如果接口声明中提供了extends子句，那么该接口就继承了父接口的方法和常量。被继承的接口称为声明接口的直接父接口。
任何实现该接口的类，必须实现该接口继承的其他接口。

13.2.3 JDK的Comparable接口

有时需要比较二个对象，但不同类型对象的比较具有不同的含义，因此Java定义了Comparable接口。因此，任何需要比较对象的类，都要实现该接口。

```
 Cloneable、Runnable、Comparable等接口均在包java.lang中：
      package java.lang;
 public interface Comparable{
              public int compareTo(Object o);
      }
```

CompareTo判断this对象相对于给定对象o的顺序，当this对象小于、等于或大于给定对象o时，分别返回负数、0或正数

例如：

```
public class ComparableRectangle extends Rectangle implements Comparable {
    /** Construct a ComparableRectangle with specified properties */
    public ComparableRectangle(double width, double height) {
             super(width, height);
    }
    /** Implement the compareTo method defined in Comparable */
    public int compareTo(Object o) {
        if (this.getArea( ) >((ComparableRectangle)o).getArea()) return 1;
        else if (this.getArea( ) <((ComparableRectangle)o).getArea()) return -1;
        else return 0;
    }
}

ComparableRectangle rec1 = new ComparableRectangle（1.0,1.0）;
ComparableRectangle rec2 = new ComparableRectangle（2.0,2.0）;
rec1.compareTo(rec2); //rec1就是this对象
```

有了Comparable接口，我们可以实现很通用的类来比较对象，例如实现一个从两个对象中找出最大者的方法。

```
public class Max{
    public static Comparable findMax (Comparable o1, Comparable o2){
```

```
        if(o1.CompareTo(o2) > 0 )
            return o1;
        else
            return o2;
    }
}
```

- 注意findMax方法的参数类型和返回类型都是Comparable（只要是实现了Comparable接口的对象都可以传进来。
  Comparable接口描述了可以比较大小的能力，一个类实现了这个接口，意味着这个类的对象直接可以比较大小）
- Max.findMax与Comparable接口的具体实现子类无关（**只和接口有关，基于接口的编程**）。只要是实现了Comparable接口的具体类的二个对象（注意是同一个具体类的二个对象）传进来，
  Max.findMax都能工作。这就是接口的好处。（程序存在的问题：如果是2个实现了Comparable接口的不同具体类对象传进来怎么办？最好通过**泛型**解决）
- 另外要注意的是：o1.CompareTo(o2)调用是动态绑定（多态）（调用具体子类对象的CompareTo方法）

```
public class ComparableRectangle extends Rectangle implements Comparable {
    /** Construct a ComparableRectangle with specified properties */
    public ComparableRectangle(double width, double height) {
             super(width, height);
    }
    /** Implement the compareTo method defined in Comparable */
    public int compareTo(Object o) {
        if (this.getArea( ) >((ComparableRectangle)o).getArea()) return 1;
        else if (this.getArea( ) <((ComparableRectangle)o).getArea()) return -1;
        else return 0;
    }
}
//注意由于篇幅所限没有用instanceOf检查o的类型。但如果o不是ComparableRectangle类型怎么办？
//这时返回什么样的整数都不合适，这个问题最好的解决办法是用泛型。
```

- 对于ComparableRectangle的两个对象r1和r2，直接调用Max.findMax(r1,r2)找出最大的对象
- 对于实现了Comparable接口任何类的二个对象（同一个类）(不管其具体实现是什么)a1和a2，都可以调用Max.findMax(a1,a2)找出最大的对象。这就是接口和多态的威力。

13.2.4 Cloneable 接口

`Cloneable` 是 Java 中的一个**标记接口（Marker Interface）**，用于指示一个类的对象可以被**克隆（clone）**。它本身不包含任何方法，但它的存在告诉 JVM 该类的实例可以使用 `Object.clone()` 方法进行复制。

---

1. Cloneable 接口的作用

- **标记作用**：`Cloneable` 只是一个标记，表示该类支持克隆。如果一个类实现了 `Cloneable`，那么它的实例可以调用 `Object.clone()` 方法进行复制。
- **安全性**：如果没有实现 `Cloneable` 接口，直接调用 `clone()` 会抛出 `CloneNotSupportedException`。

---

2. 如何使用 Cloneable？

（1）基本使用

要让一个类支持克隆，需要：

1. **实现 `Cloneable` 接口**（否则 `clone()` 会抛出异常）。
2. **重写 `Object.clone()` 方法**（因为 `Object.clone()` 是 `protected` 的，需要提升为 `public`）。

示例代码

```
class Person implements Cloneable {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public Person clone() throws CloneNotSupportedException {
        return (Person) super.clone(); // 调用 Object.clone()
    }

    @Override
    public String toString() {
        return "Person{name='" + name + "', age=" + age + "}";
    }
}

public class Main {
    public static void main(String[] args) throws CloneNotSupportedException {
        Person p1 = new Person("Alice", 25);
        Person p2 = p1.clone(); // 克隆 p1

        System.out.println(p1); // Person{name='Alice', age=25}
        System.out.println(p2); // Person{name='Alice', age=25}
    }
}
```

（2）注意事项

- `Object.clone()` **是浅拷贝（Shallow Copy）：**
  - 如果类中有引用类型（如 `String[]`、`List` 等），克隆后的对象会共享这些引用，可能导致数据不一致。
  - 如果需要**深拷贝（Deep Copy）**，必须手动复制引用对象。

浅拷贝 vs 深拷贝

类型	特点	示例
浅拷贝	基本类型直接复制，引用类型只复制引用（新旧对象共享引用）。	`super.clone()`
深拷贝	引用类型也创建新对象，新旧对象完全独立。	手动复制引用对象

深拷贝示例

```
class Student implements Cloneable {
    private String name;
    private int[] scores; // 引用类型

    public Student(String name, int[] scores) {
        this.name = name;
        this.scores = scores;
    }

    @Override
    public Student clone() throws CloneNotSupportedException {
        Student cloned = (Student) super.clone(); // 浅拷贝
        cloned.scores = scores.clone(); // 手动深拷贝数组
        return cloned;
    }
}
```

---

3. 为什么 Cloneable 是一个标记接口？

- **历史原因**：Java 早期没有注解（Annotation），所以用标记接口表示某种能力。
- `clone()` **方法在** `Object` **类中：**
  - `Object.clone()` 是 `protected` 的，需要子类重写并提升为 `public`。
  - 如果没有 `Cloneable` 接口，直接调用 `clone()` 会抛出异常。

---

4. 最佳实践

1. **尽量使用** `Cloneable` **进行对象复制**，而不是手动 `new` + `set`（性能更高）。
2. **引用类型要深拷贝**，避免共享数据。
3. **考虑使用** `Copy Constructor` **或** `Serialization` **替代** `clone()`（更灵活）：

```
// 使用 Copy Constructor
public Person(Person original) {
    this.name = original.name;
    this.age = original.age;
}
```

---

5. 总结

要点	说明
`Cloneable` 的作用	标记类支持克隆，否则 `clone()` 会抛出异常。
`clone()` 方法	默认是浅拷贝，引用类型需要手动深拷贝。
替代方案	`Copy Constructor`、`Serialization` 更灵活。

推荐：如果只是简单复制，用 `Cloneable`；如果需要复杂控制，用 `Copy Constructor` 或工具库（如 Apache Commons `SerializationUtils`）。

13.3 接口和抽象类的比较

特性	接口（Interface）	抽象类（Abstract Class）
多重继承	一个接口可以继承多个接口（`extends A, B`）。	一个类只能继承一个抽象类（单继承）。
方法实现	不能提供任何代码（Java 8前）；默认方法（`default`）和静态方法（`static`）可提供实现。	非抽象方法可以提供完整代码；抽象方法需用 `abstract` 显式声明。
数据字段	只包含 `public static final` 常量，必须在声明时初始化。	可包含实例变量、静态变量、实例/静态常量（非常量字段可修改）。
设计目的	描述类的**外围能力**（如 `Runnable`），体现“can-do”关系（`instanceof` 成立）。	定义类的**核心特征**（如 `Person`），体现“is-a”关系（`instanceof` 成立）。
简洁性	常量和方法默认修饰符可省略（`public static final` 和 `public abstract`）。	需显式声明抽象方法（`abstract`）；可包含共享代码。
添加新方法的影响	新增方法需在所有实现类中强制实现（除非是 `default` 方法）。	可为新方法提供默认实现，现有代码无需修改。
适用场景	需要多继承或定义行为契约时（如 `Comparable`）。	需要复用代码或定义类层次结构时（如模板方法模式）。

补充说明：

4. **接口的默认方法**（Java 8+）：

```
interface A {
    default void foo() { System.out.println("A"); }
}
```

- 解决接口扩展性问题，避免破坏现有实现类。

5. **抽象类的常量与变量：**

```
abstract class B {
    static final int MAX = 100; // 常量
    int count;                  // 实例变量（可修改）
}
```

6. **多重继承冲突：**
   - 接口：若多个父接口有同名默认方法，子接口需重写或指定（`A.super.foo()`）。
   - 抽象类：无此问题（单继承）。

13.4 包装类提供的接口和方法

在 Java 中，**包装类**用于将**基本数据类型（Primitive Types）封装为对象（Object）**。它们位于 `java.lang` 包中，主要用于以下场景：

7. **泛型：**泛型只能使用对象类型（如 `List<Integer>`）。
8. **集合框架：**集合类（如 `ArrayList`、`HashMap`）只能存储对象。
9. **方法参数：**某些方法需要对象类型（如 `Object`）。
10. **工具方法：**包装类提供了许多实用的静态方法（如 `Integer.parseInt()`）。

---

1. 包装类的分类

Java 为每种基本数据类型提供了对应的包装类：

基本类型	包装类
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`

---

2. 包装类的核心功能

（1）基本类型与包装类的转换

- **装箱（Boxing）**：将基本类型转换为包装类。

```
int i = 10;
Integer integer = Integer.valueOf(i); // 手动装箱
Integer autoBoxed = i; // 自动装箱（Java 5+）
```

- **拆箱（Unboxing）**：将包装类转换为基本类型。

```
Integer integer = 10;
int i = integer.intValue(); // 手动拆箱
int autoUnboxed = integer; // 自动拆箱（Java 5+）
```

（2）字符串与基本类型的转换

- **字符串转基本类型**：

```
int i = Integer.parseInt("123"); // 字符串转 int
int i = Integer.parseInt(“11”,2); //3
int i = Integer.parseInt(“12”,8); //10
int i = Integer.parseInt(“1A”,16); //26
double d = Double.parseDouble("3.14"); // 字符串转 double
```

- **基本类型转字符串**：

```
String s1 = Integer.toString(123); // int 转字符串
String s2 = 123 + ""; // 通过字符串拼接
```

（3）常量与方法

- **常量**：

```
System.out.println(Integer.MAX_VALUE); // 2147483647
System.out.println(Double.NaN); // NaN（非数字）
```

- **方法**：

```
Integer i = 10;
System.out.println(i.compareTo(5)); // 1（i > 5）
System.out.println(Double.isNaN(0.0 / 0.0)); // true
```

---

3. 包装类的常用方法

（1）`valueOf()`

- 将基本类型或字符串转换为包装类。

```
Integer i = Integer.valueOf(10); // 基本类型转包装类
Integer j = Integer.valueOf("10"); // 字符串转包装类
```

（2）`parseXxx()`

- 将字符串转换为基本类型。

```
int i = Integer.parseInt("10"); // 字符串转 int
double d = Double.parseDouble("3.14"); // 字符串转 double
```

（3）`toString()`

- 将基本类型或包装类转换为字符串。

```
String s1 = Integer.toString(10); // int 转字符串
String s2 = Integer.valueOf(10).toString(); // 包装类转字符串
```

（4）`compareTo()`

- 比较两个包装类的大小。

```
Integer i = 10;
System.out.println(i.compareTo(5)); // 1（i > 5）
System.out.println(i.compareTo(10)); // 0（i == 10）
System.out.println(i.compareTo(15)); // -1（i < 15）
```

（5）`equals()`

- 比较两个包装类的值是否相等。

```
Integer i = 10;
Integer j = 10;
System.out.println(i.equals(j)); // true
```

（6）`isNaN()`（仅 `Float` 和 `Double`）

- 判断是否为非数字（NaN）。

```
System.out.println(Double.isNaN(0.0 / 0.0)); // true
```

---

4. 注意事项

1. **缓存机制：**
    - `Integer`、`Byte`、`Short`、`Long`、`Character` 对部分值（如 `-128` 到 `127`）有缓存，`valueOf()` 会返回缓存对象。

```
Integer a = 127;
Integer b = 127;
System.out.println(a == b); // true（缓存对象）
Integer c = 128;
Integer d = 128;
System.out.println(c == d); // false（新建对象）
```

2. **空指针异常：**
    - 自动拆箱时，如果包装类为 `null`，会抛出 `NullPointerException`。

```
Integer i = null;
int j = i; // NullPointerException
```

3. **性能开销：**
    - 包装类是对象，比基本类型占用更多内存，操作也更慢。在性能敏感的场景中，优先使用基本类型。

---

5. 总结

功能	示例
装箱/拆箱	`Integer i = 10;`（自动装箱），`int j = i;`（自动拆箱）
字符串转基本类型	`int i = Integer.parseInt("10");`
基本类型转字符串	`String s = Integer.toString(10);`
比较大小	`i.compareTo(5)`
判断相等	`i.equals(j)`
判断 NaN	`Double.isNaN(0.0 / 0.0)`

第15章 事件驱动编程和动画

15.1 引言

略

15.2 事件和事件源

略

15.3 事件处理器和处理事件

略

第19章 泛型

19.1 基本概念

19.1.1 泛型

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**编译时而不是运行时**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。

```
public class GenericDemo1 {
    //定义泛型函数，类型参数为T（代表某一种类型），T为类型形参
    public static <T> add(T value1, T value2){ return value1 + value2;}
}
//调用泛型函数，需要给出类型实参
GenericDemo1.<Integer>add（1,2）；//显示地给出类型实参为Integer，传递给形参T。
GenericDemo1.add(1,2) //编译器自动可以推断出T为Integer（类型推断）
```

泛型类（Generic Class）是带**形式化类型参数**的类。形式化类型参数是一个逗号分隔的变量名列表，位于类声明中类名后面的尖括号<>中。下面的代码声明一个泛型类Wrapper，它接受一个形式化类型参数T：

```
public class Wrapper<T>  {

}
```

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<String>，Wrapper<Integer>，这些参数化类型是泛型类Wrapper<T>的实例类型（类似于Circle类型有实例对象c1,c2）

```
Wrapper<String>  stringWrapper = new Wrapper<String>();
Wrapper<Circle>  circleWrapper = new Wrapper<Circle>();
```

> 参数化类型(Parameterized Types)是在JLS里面使用的术语，为了方便描述，本章后面称为实例类型

19.1.2 Class类和Class对象

要理解RTTI在Java中的工作原理，就必须知道类型信息在运行时是如何表示的。
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。
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。

1. 如何获取Class对象

(1)**Class.forName**方法，通过类的完全限定名字符串获取Class对象。是Class类的静态方法

```
public class ClassDemo {
    public static void main(String[] args){
        try {
            Class clz = Class.forName("ch13.Manager"); //参数是类完全限定名字符串
            System.out.println(clz.getName()); //产生完全限定名ch13.Manager
            System.out.println(clz.getSimpleName()); //产生简单名Manager

            Class superClz = clz.getSuperclass(); //获得直接父类型信息
            System.out.println(superClz.getName()); //产生完全限定名ch13.Employee
            System.out.println(superClz.getSimpleName()); //产生简单名Employee
        } catch (ClassNotFoundException e) {
```

```
            e.printStackTrace();
        }
    }
}
```

> 注意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)利用类字面常量：类名.class，得到类对应的Class对象

```
public class ClassDemo {
    public static void main(String[] args){
        Class clz = Manager.class;  // Manager.class得到Manager的Class对象.赋给引用clz
        System.out.println(clz.getName()); //产生完全限定名ch13.Manager
        System.out.println(clz.getSimpleName()); //产生简单名Manager
    }
}
```

- 类字面常量不仅可以用于类，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
- 某个类名.class是Class类型的字面量
  正如int类型的字面量有1，2，3，
  Class类型的字面量有Person.class，Employee.class, Manager.class，它们都是Class类型的实例

(3)通过对象。如果获得一个对象的引用o，通过o.getClass()(Object类的final方法)方法获得这个对象的类型的Class对象

```
public class ClassDemo {
    public static void main(String[] args){
        Object o = new Manager();
        Class clz = o.getClass();
        System.out.println(clz.getName()); //产生完全限定名ch13.Manager
        System.out.println(clz.getSimpleName()); //产生简单名Manager
    }
}
```

> 注意：getClass返回的是运行时类型

2.泛型Class引用

```
//非泛型的Class引用（即不带类型参数的Class引用）可指向任何类型的Class对象，但这样不安全
Class clz ;     //注意警告， Class is a raw type. References to generic type Class<T>
should be parameterized
clz= Manager.class; //OK
clz = int.class; //OK

//有时我们需要限定Class引用能指向的类型：加上<类型参数>。这样可以强制编译器进行额外的类型检查
Class<Person> genericClz;    //泛型Class引用，Class<Person>只能指向Person的类型信息，
<Person>为类型参数
```

```
genericClz = Person.class;  //OK
//genericClz = Manager.class;
//Error，不能指向非Person类型信息。注意对于类型参数，编译器检测时不看继承关系。

//能否声明一个可用指向Person及其子类的Class对象的引用？为了放松泛型的限制，用通配符?表示任何类型，并且与extends结合，创建一个范围
Class<? extends Person> clz2;  //引用clz2可以指向Person及其子类的类型信息
clz2 = Person.class;
clz2 = Employee.class;
clz2 = Manager.class;
//注意Class<?> 与Class效果一样，但本质不同，一个用了泛型，一个没有用泛型。 Class<?>等价于Class<? extends Object >
```

19.1.3 反射

利用Class对象我们可以在运行时动态地创建对象，调用对象的方法

```
Constructor[] ctors = clz.getConstructors();//通过 clz可以得到这个类的所有构造函数对象
Method[] methods = clz.getMethods(); //会显示所有方法，包括继承的
Method[] methods = clz.getDeclaredMethods(); //本类定义的方法
```

```
public class ReflectDemo {
    public static void main(String[] args) {
        try {
            Class clz = Class.forName("ch13.Student");
              //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
              Student s2 =
(Student)clz.getConstructor(String.class).newInstance("John");
              //invoke method
              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();
        }
   }
}
Name:Marry
Name:John
```

- 首先得到参数类型为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

- clz.getMethod(“setName”,
  String.class);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

19.2 动机和优点

对非泛型类，保证放进去对象的类型一致性变成了程序员的责任。
而泛型，保证放进去对象的类型一致性是编译器的责任，编译器是不会犯错的
例：

```
public class ArrayList<E> extends AbstractList<E>
    implements List<E>,
    RandomAccess, Cloneable, java.io.Serializable{

    // 内部就是一个Object[]数组
    transient Object[] elementData;

    E elementData(int index) {
        return (E) elementData[index];
    }

    public boolean add(E e) {
        ensureCapacityInternal(size + 1);
        elementData[size++] = e;
        return true;
    }
    public E get(int index) {
        rangeCheck(index);//检查index是否越界
        return elementData(index);
    }
    //其它代码
}
```

ArrayList<E>定义了一个带类型形参的泛型类, 类型参数E是形参

ArrayList<String>
是一个参数化类型(**实例类型**)，其中String作为一个具体类型（实参）传递给形参E。
这里借用了术语"实例"，不是指对象，而是一个具体的类型。

特别重要的是：类型实参String传递给类型形参E是发生在**编译时**（不是运行时）。因此，对于下面的语句，编译器会用String代替E，对代码进行类型检查（意思编译前一页的代码时，E全部换成String）。

ArrayList<String> list = new ArrayList<>();//用实例类型ArrayList<String> 声明引用变量list
list.add("China");//编译器会根据类型实参String检查传入add方法的对象类型是否匹配，否则报错

- 泛型类型的参数实参**必须是引用类型**

19.3 定义泛型类和接口

例：用泛型定义栈类

```
import java.util.ArrayList;
public class GenericStack<E> {
         private ArrayList<E> list = new ArrayList<E>();
         public boolean isEmpty() {
        return list.isEmpty();
         }
```

```
        public int getSize() {
      return list.size();
        }
        public E peek() {
       return list.get(getSize() - 1);//取值不出栈
        }
        public E pop() {
       E o = list.get(getSize() - 1) ;
       list.remove(getSize() - 1);
       return o;
         }

public void push(E o) {
       list.add(o);
           }
        public String toString() {
       return "stack: " + list.toString();
         }
}
```

注意：

GenericStack<E>构造函数形式是擦除参数类型后的GenericStack( ),不是GenericStack<>();

```
public class GenericContainer<E> {

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

19.4 泛型方法

除了可以定义泛型接口和泛型类，也可以定义泛型方法。下面的例子在一个非泛型类里定义了泛型方法

```
public class GenericMethodDemo {
    public static void main(String[] args) {
        Integer[] integers = {1,2,3,4,5};
        String[] strings = {"Londen","Paris","New York","Austin"};
        GenericMethodDemo.<Integer>print(integers);
        GenericMethodDemo.<String>print(strings);
    }
      //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

    public  static <E> void print(E[] list){
        for(int i = 0 ; i <list.length; i++){
            System.out.print(list[i]+" ");
            System.out.println();
                    //声明泛型方法，将类型参数<E>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

        }
    }
}
```

在定义泛型类、泛型接口、泛型方法时，可以将**类型参数**指定为另外一种类型（或泛型）的子类型（用extends），这样的类型参数称之为**受限的**（bounded）
想实现泛型方法比较二个几何对象的面积是否相等，几何对象类型很多，都从GeometricObject派生

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

注意:
　　类型参数放置的位置，应放在方法的返回类型之前(定义泛型方法)或者类名之后（定义泛型类时）

19.5 原始类型和向后兼容

- 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。如：Max.findMax(“Welcome”,new Integer(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
```

> 这个时候语句Max.findMax(“Welcome”, new Integer(123)）；会引起编译时错误，因为findMax方法要求两个参数类型必须一致，且E必须实现Comparable<E> 接口

19.6 通配泛型

19.6.1 问题引入

```
public class WildCardNeedDemo {
    public static double max(GenericStack<Number> stack){
        double max = stack.pop().doubleValue();
        while (! stack.isEmpty()){
            double value = stack.pop().doubleValue();
            if(value > max)
                max = value;
        }
        return max;
    }


    public static void main(String[] args){
        GenericStack<Integer> intStack = new GenericStack<>();
        intStack.push(1);intStack.push(2);intStack.push(3);
        System.out.println("Th max value is " + max(intStack));
             //出错，因为intStack不是GenericStack<Number>实例
    }
}
```

- Integer是Number的子类，但是GenericStack<Integer>并不是GenericStack<Number>的子类。
  原因：泛型集合类型没有协变性
  苹果是水果的子类型，但是装满苹果的篮子不是装满水果的篮子的子类型。前面说过，编译器在编译时，是不考虑类型实参之间的继承关系的。
  如何解决？
  使用通配泛型
  `double max(GenericStack<? extends Number> stack)`
  extends表示了类型参数的范围关系。
  `GenericStack<? extends Number>`才是GenericStack<Integer> 的父类，
  GenericStack<Number>不是GenericStack<Integer> 的父类

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的父类型，下界通配符，T定义了类型下限

19.6.2 协变性

1. 数组的协变性（Covariant）

数组的协变性是指：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Friut[]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（扩展到任何泛型类）没有协变性**

- 因为：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，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类型错误。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

19.6.3 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
```

采用上界通配泛型 ？ extends

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（有多态性）
Fruit f = list.get(0);
```

因为ArrayList<? extends Fruit>意味着该list集合中存放的都是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

假设已经实例化好了另外一个容器，对象已经放入其中，这时用ArrayList<? extends Fruit> list指向这个另外的容器，那么我们可以通过list取出容器的所有对象而没有任何问题

```
ArrayList<Apple> apples = new ArrayList<Apple>();
//调用apples.add方法添加很多Apple及其子类对象

ArrayList<? extends Fruit> list = apples; //现在ArrayList<? extends Fruit>类型的引用指向apples
for（int i = 0; i < list.size(); i++）{
    Fruit f = list.get(i);
//运行时从容器里取出的都是Apple及其子类对象，赋值给Fruit引用没问题
}
```

这个例子还是比较极端（纯粹是语法功能演示），实际更有意义的是作为方法参数：该方法接受一个放好对象的容器，然后在方法里只是逐个取出元素进行处理

```
public static void handle(ArrayList<? extends Fruit> list){ //注意方法里只能从list get元素
      for(int i = 0; i < list.size(); i++){
            Fruit o = list.get(i); //可以确定list里面对象一定是Fruit或子类类型
            //处理对象o，注意这时调用o的实例方法时具有多态性
    }
}

ArrayList<Apple> appleList = new ArrayList<>(); //等价于new ArrayList<Apple>();
appleList.add(new Apple());//ArrayList<Apple>是具体类型，编译器很清楚地知道类型参数是Apple这时可以add
//由于形参类型ArrayList<? extends Fruit>是实参类型ArrayList<Apple>的父类型，因此实参可以传给形参
handle(appleList);
类似地，我们可以实例化装满Orange的篮子(ArrayList<Orange>对象),装满Orange，传入handle方法。这就是ArrayList<? extends Fruit> list的真正意义：作为方法参数。
```

19.6.4 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T的父类型为类型参数的实例类型
ArrayList<? super Fruit> list = new ArrayList<Fruit>(); //这时new后边的Fruit可以省略
ArrayList<? super Fruit> list2 = new ArrayList<Object>(); //允许，Object是Fruit父类
ArrayList<? super Fruit> list3 = 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
```

> 为什么可以加入Fruit及其子类对象？因为编译时编译器知道list里面能放入的对象类型是Fruit及其父类型（至少是Fruit类型），而实际加入的对象的类型Fruit、Apple、 Jonathan、Orange都是Fruit的子类型，当然也是Fruit及其父类型，当然可以加入。但Object就不行（因为list里面能放入的对象类型是Fruit及其父类型，但只能是一种类型，不一定是Object，可能是Fruit）
> 或者这样理解：list.add的形参类型为Fruit（或其父类），当然可以加入Apple

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）

19.6.5 ？extends 和？ super的理解

```
//先看看通配泛型 ？ 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Array
List<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*/

//? 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，这里面又分二种情况
    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，这里面又分二种情况
    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（
list引用可以指向ArrayList<Fruit>|
ArrayList<Object>，因此从篮子里拿出来对象解释成Object最安全 ）

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
```

19.7 泛型擦除和对泛型的限制

19.7.1 概念

- 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。

```
public class TypeErasureTest {
    public static void main(String[] args){
        ArrayList<String> strList = new ArrayList<>();
        ArrayList<Fruit> fruitList = new ArrayList<>();
        Class clz1 = strList.getClass(); //getClass返回运行时信息
        Class clz2 = fruitList.getClass();
        System.out.println(clz1.getSimpleName());   //ArrayList
        System.out.println(clz2.getSimpleName());   //ArrayList
        System.out.println(clz1 == clz2);           //true
    }
}
```

所有参数化类型（实例类型）ArrayList<String> 、ArrayList<Fruit>
在运行时共享同一个类型：ArrayList。
请大家再回到PPT第2页去理解最后一段话

- 泛型类会擦除类型参数，所有泛型的实例类型共享擦除后形成的原始类型如ArrayList
    - 泛型类所有实例类型在运行时共享原始类型，如：
      ArrayList<String> list1 = new ArrayList<>( );
      ArrayList<Integer> list2= new ArrayList<>( );
    - 在运行时只有一个擦除参数类型后的原始ArrayList类被加载到JVM中
      所以， list1 instanceOf ArrayList<String>是错误的，可用：
      list1 instanceOf ArrayList
      list2 instanceOf ArrayList
      instanceOf是根据运行时类型进行检查
- 使用泛型类型的限制
    - 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[cpacity]; //错误
      new是运行是发生的，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[cpacity];
      //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

19.7.2 例：一维数组的泛型包装类

```
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
    }
}
```

利用反射机制改进

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

    public GenericOneDimensionArray(Class<? extends T> clz,int size){
        elements = (T[])Array.newInstance(clz,size);
    }

    //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 使用泛型类型的限制

使用泛型类型的限制：不能new泛型数组（数组元素不能是泛型），但可以声明

- 不能使用new A<E>[ ]的数组形式，因为E已经被擦除
  ArrayList<String>[ ] list = new ArrayList<String>[10]；//错误
- E已经被擦除，只能用泛型的原始类型初始化数组, 必须改为new ArrayList[10]
  ArrayList<String> [ ] list = new ArrayList[10]；
    - 为什么这里不需要强制类型转换：参数化类型与原始类型的兼容性

- 参数化类型对象可以被赋值为原始类型的对象，原始类型对象也可以被赋值为参数化类型对象
  ArrayList a1 = new ArrayList(); //原始类型
  ArrayList<String> a2 = a1; //参数化类型

- **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。

- 异常类不能是泛型的。泛型类不能继承java.lang.Throwable。

19.8 如何实现带泛型参数的对象工厂

```
public class ObjectFactory<T> {
	private Class<T> type;  //定义私有数据成员，保存要创建的对象的类型信息
	public ObjectFactory(Class<T> type) {//构造函数传入要创建的对象的类型信息
		this.type = type;
	}
	public T create() {//对象工厂的create方法负责产生一个T类型的对象，利用newInstance
		T o = null;
		try {
			o= type.newInstance();
		} catch (InstantiationException | IllegalAccessException e) {
			e.printStackTrace();
		}
		return o;
	}
}

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

第30章 多线程和并行程序设计

30.1 线程的概念

30.1.1 进程与线程

“程序”代表一个静态的对象，是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”代表一个动态的对象，是程序的一个执行过程，存在于系统的内存中。一个进程对应于一个程序
“线程”是运行于某个进程中，用于完成某个具体任务的顺序控制流程，有时被称为轻型进程。

- 但是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
- 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
- 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。
- 线程：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。
- 多线程共享一个CPU：某时刻，只能有一个线程在使用CPU

现代OS都将**线程作为最小调度单位**，进程作为资源分配的最小单位。分配给进程的资源（如文件，外设）可以被进程里的线程使用

30.1.2 线程的作用

- 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。
- 特别是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
- 一个典型的GUI程序分为
  - 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接口和线程类Thread

30.2.1 线程的创建

创建线程方法：线程的执行逻辑（后面叫线程任务）必须实现java.lang.Runnable接口的唯一run方法。此外，由于Thread实现了Runnable接口，也可以通过Thread派生线程类。
因此有两种方法可以实现同一个或多个线程的运行：
(1) **定义Thread类的子类并覆盖run方法；**
(2) **实现接口Runnable的run方法。**

1. 通过实现Runnable接口创建线程

- 实现Runnable接口，需要实现唯一的接口方法run
- void run( )
- 该方法定义了线程执行的功能
- 创建实现Runnable接口的类的对象
- 利用Thread类的构造函数创建线程对象
  public Thread (Runnable target)
- 通过线程对象的start()方法启动线程

```
//Custom task class
class TaskClass implements Runnable {
   ... //可以有自己的数据成员
   public TaskClass(...) {
      ...
   }

   //Implement the run method in Runnable
   public void run() {
      //Tell system how to run custom thread
      ...
   }
}

//Client Class
public class Client {
   ...
   public void someMethod(...) {
   ...
   // Create an instance of TaskClass
   Runnable task = new TaskClass(...);

   // Create a thread
   Thread thread = new Thread(task);

   // Start a thread
   thread.start(); // 启动后自动执行task.run
   }
}
```

1. 通过线程任务类（TaskClass）创建任务对象（task）
2. 以任务对象task为参数new
   Thread对象。Thread对象代表一个线程，线程的执行内容由任务对象task定义。
3. 通过线程对象thread启动线程thread.start(
)，任何线程只能启动一次，多次调用产生IllegalThreadStateException异常。

例子:程序创建并运行二个线程：第一个线程打印100次字母a；第二个线程打印100次字母b

```
class PrintChar implements Runnable //实现Runnable接口
{
      private char charToPrint;  // The character to print
      private int times;  // The times to repeat
      public PrintChar(char c, int t){  charToPrint = c;  times = t; }
      public void run(){ //实现Runnable中声明的run方法
          for (int i=1; i < times; i++) System.out.print(charToPrint);
      }
}

public class RunnableDemo
{
    public static void main(String[] args){
        //以PrintChar对象实例为参数构造Thread对象
        Thread printA = new Thread(new PrintChar('a',100));
        Thread printB = new Thread(new PrintChar('b',100));
        printA.start();
        printB.start();
    }
}
```

2. 通过继承Thread类创建线程

```
// Custom thread class
class CustomThread extends Thread {
  //数据成员
  public CustomThread(...) {
    ...
  }

  public void run() {
    // Tell system how to perform this task
    ...
  }
  ...
}
//Client class
public class Client {
  ...
  public void someMethod() {
    ...
    // Create a thread
    Thread thread1 = new CustomThread();
    // Start thread
    thread1.start( ); //激活thread1对象的run

    // Create a thread
    Thread thread2 = new CustomThread( );
    // Start thread
    thread2.start(); //激活thread2对象的run
  }
  ...
}
```

1. 定义Thread类的扩展类（CustomThread）

2. 通过扩展类（CustomThread）创建线程对象（thread）
3. 通过线程对象thread启动线程thread.start()
4. 线程和线程任务混在一起，**不建议使用**,Java不支持多继承，CustomThread继承了Thread类不能再继承其他类

例子:

```
class PrintChar extends Thread //继承Thread类
{
	private char charToPrint;  //要打印的字符
	private int times;  //打印的次数
	public PrintChar(char c, int t){ charToPrint = c; times = t; }
	public void run() {//覆盖run方法，定义线程要完成的功能
		for (int i=1; i < times; i++)
			System.out.print(charToPrint);
	}
}
public class ThreadDemo {
	public static void main(String[] args) {
	    Thread printA = new PrintChar('a',100); //创建二个线程对象
	    Thread printB = new PrintChar('b',100);
	    printA.start(); //启动线程
	    printB.start(); //启动另外一个线程
	}
}
```

30.2.2 线程的状态切换

下面一些情况导致线程从运行状态转到阻塞状态：
1：调用了sleep
2：调用了Object wait(
)方法、条件对象的await方法，Thread的join方法以等待其他线程，或者等待资源锁
3：发出了阻塞式IO操作请求，并等待IO操作结果（如等待阻塞式Socket的数据到来）

线程由阻塞状态被唤醒后，回到就绪态。唤醒的原因
1：sleep时间到
2：调用wait（await）的线程被其他线程notify，调用join方法的线程等到了其他线程的完成，线程拿到了资源锁
3：阻塞IO完成

30.2.3 线程的优先级

- 线程优先级范围从1－10，数字越高越能被优先执行。但优先级高并不代表能独自占用执行时间片，可能是优先级高得到越多的执行时间片，反之，优先级低的分到的执行时间少但不会分配不到执行时间
- 每个线程创建时赋予默认的优先级Thread.NORM_PRIORITY.
- 通过setPriority(int priority)为线程指定优先级.
- 用getPriority()方法获取线程的优先级.

- JAVA定义的优先级：1～10
- Thread类有int 类型的常量：
    - Thread.MIN_PRIORITY（1）
    - Thread.MAX_PRIORITY（10）
    - Thread.NORM_PRIORITY（5）
- 多个线程只能是“宏观上并行，微观上串行”
- 在有限个CPU的系统中确定多个线程的执行顺序称为线程的调度
- 自私的线程

```
    run() {
                while (true) {
                }
          }
```

应适当地在run()里sleep或yield一下，让其他线程有更多机会被运行。
不要编写依赖于线程优先级的程序

30.2.5 线程类Thread的yield，sleep方法

使用 yield() 方法为其他线程让出CPU时间：

```
public void run() {
    for (int i = 1; i < times; i++) {
        System.out.print(charToPrint);
        Thread.yield();  //挂起进入ready，给其它进程调度机会
     }
}
```

sleep(long mills)方法将线程设置为休眠状态，确保其他线程执行:

```
public void run() {
    try {//循环中使用sleep方法，循环放在try-catch块中
        for (int i = 1; i < times; i++) {
            System.out.print(charToPrint);
            if (i >= 50) Thread.sleep(1);
        }
    }
    // 必检异常：其它线程调当前线程（正在休眠）interupt方法会抛出该异常
    catch (InterruptedException ex { }
}
```

> 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interupt方法被其它线程调用），会抛出InterruptedException

30.2.6 线程类Thread的-join方法

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printA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printA执行完毕，main线程被唤醒执行printB.start();

运行结果是全部a打印完才开始打印b

```
public class JoinDemo {
    public static void main(String[] args) throws InterruptedException{
        Thread printA = new Thread(new PrintChar('a',100));
        Thread printB = new Thread(new PrintChar('b',100));
        printA.start();  //在主线程里首先启动printA线程
        printA.join(); //主线程被阻塞，等待printA执行完
        printB.start(); //主线程被唤醒，启动printB线程
    }
}
class PrintChar implements Runnable
{
    private char charToPrint;  // The character to print
    private int times;  // The times to repeat
    public PrintChar(char c, int t){  charToPrint = c;  times = t; }
    public void run(){ //实现Runnable中声明的run方法
        for (int i=1; i < times; i++)
            System.out.print(charToPrint);
    }
}
```

在线程任务对象print100的run中启动新线程Thread4，并调用Thread4 的join( )
方法，，等待Thread4结束:

```
class PrintNum implements Runnable{ //实现新的线程任务类，打印数字
    private int lastNum;
    public PrintNum(int n){ lastNum = n; }
    @Override
    public void run() {
       Thread thread4=new Thread(new PrintChar('c',40));
        thread4.start();
        try{
            for(int i=1;i<lastNum;i++){
                System.out.print(" " + i);
                if(i == 50) thread4.join(); //join方法可以给参数指定至多等若干毫秒
            }
        }
        catch(InterruptedException e){ } //join方法可能会抛出这个异常
        }
}
```

启动tPrint100线程：

```
Runnable print100 = new PrintNum(100); //线程任务对象
Thread tPrint100 = new Thread(print100); //线程对象
tPrint100.start();
```

30.3 线程池

由于要为每一个线程任务创建一个线程（Thread对象），对于有大量线程任务的场景就不够高效（当线程任务执行完毕，即run方法结束后，Thread对象就消亡，然后又为新的线程任务去new新的线程对象…, 当有大量的线程任务时，就不断的new Thread对象，Thread对象消亡，再new Thread对象…）
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” 线程对象来提高性能.
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
例：

```
import java.util.concurrent.*;

public class ExecutorDemo {
  public static void main(String[] args) {
    // Create a fixed thread pool with maximum three threads
    ExecutorService es= Executors.newFixedThreadPool(3);

    // Submit runnable tasks to the executor
       es.execute(new PrintChar('a', 100));
       es.execute(new PrintChar('b', 100));
       es.execute(new PrintNum(100));

    // Shut down
    es.shutdown();
  }
}
```

30.4 线程同步

- 线程同步用于协调多个线程访问公共资源
  公共资源被多个线程同时访问，可能会遭到破坏
  （程序清单30-4：AccountWithoutSync.java）
- **临界区**(critical region)：可能被多个线程同时进入的程序的一部分区域
  所以需要对临界区同步，保证任何时候只能有1个线程进入临界区
- 可以用synchronized关键字来**同步临界区**
  - 临界区可以是方法，包括静态方法和实例方法，那么被synchronized关键字修饰的方法叫同步方法
  - 临界区也可以是语句块，也可以用synchronized关键字来同步语句块：
    如synchronized(this) { …}
- m除了用synchronized关键字 ，还可利用加锁同步临界区

30.4.1 线程同步-synchronized

- `synchronized`可用于同步方法
- 使用关键字synchronized 来修饰方法：
  public synchronized void deposit(double amount)
- 一次只有一个线程可以进入这个同步方法
- synchronized关键字是如何做到方法同步的？通过**加锁**：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）
  - 锁是一种实现资源排他使用的机制
  - 对于synchronized实例方法，是对调用该方法的**对象（this对象）加锁**
  - 对于synchronized静态方法，是对拥有这个静态方法的**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

- synchronized也可以同步语句块
- 被synchronized关键字同步的语句块称为同步块(synchronized Block)
  synchronized (expr) { statements; } ，
- 表达式expr求值结果必须是一个**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（**唤醒后进入就绪态**）
- 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

30.4.2 线程同步-加锁同步

- 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（后面会展开详细讨论）
- 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

可重入性锁描述这样的一个问题：一个线程在持有一个锁的时候，它能否再次（多次）申请该锁。如果一个线程已经获得了锁，它还可以再次获取该锁而不会死锁，那么我们就称该锁为可重入锁。通过以下伪代码说明：

```
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
}
```

Java关键字synchronized隐式支持重入性

例子：

```
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
public class AccountWithSyncUsingLock {
    private static Account account = new Account();
    public static void main(String[] args) {
        ExecutorService executor = Executors.newCachedThreadPool();
        // Create and launch 100 threads
        for (int i = 0; i < 100; i++) {executor.execute(new AddAPennyTask());}
        executor.shutdown();
        // Wait until all tasks are finished
        while (!executor.isTerminated()) { }
        System.out.println("What is balance ? " + account.getBalance());
    }
    // A thread for adding a penny to the account
    public static class AddAPennyTask implements Runnable {
        public void run() {account.deposit(1); }
    }

    public static class Account {// An inner class for account，主要变化在账户类
        private static Lock lock = new ReentrantLock(); //
注意这里是静态的，被所有Account实例共享
        private int balance = 0;
        public int getBalance() {return balance;}
        public void deposit(int amount) {
            lock.lock( ); // Acquire the lock
//在这里加锁（临界区开始），第一个进入这个方法的线程获得锁，把deposite方法锁住。其他进入方法
的线程必须等待这把锁，因为进入阻塞状态

            try {
                int newBalance = balance + amount;
                Thread.sleep(5);
                balance = newBalance;
            }
            catch (InterruptedException ex) {   }
            finally { lock.unlock(); // Release the lock，在finally中进行锁的释放。}
//在finally块里释放锁，其它等待这把锁的线程被唤醒，第一个获得锁的线程可以进入该方法了，进去后
又对deposite上锁…
```

```
        }
    }
}
```

30.4.3 线程同步的几个场景
场景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**

场景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**

场景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。

场景4

30.4.4 线程同步-总结和思考
- 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）。
对于场景4，请思考，如果把ResourceWithLock里的实例成员lock改成静态成员，结果有什么不一样？

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（见ResourceWithLock的里的lock对象定义），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。

30.4.5 线程同步-线程协作

- 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
- 锁和条件是Java 5中的新内容，在Java 5之前，线程通信是使用对象的内置监视器（Object类的wait/signal/signalAll）编程实现
- 锁和条件比内置监视器更加强大且灵活，因此无须使用内置监视器，但要注意遗留代码中的内置监视器

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

例：

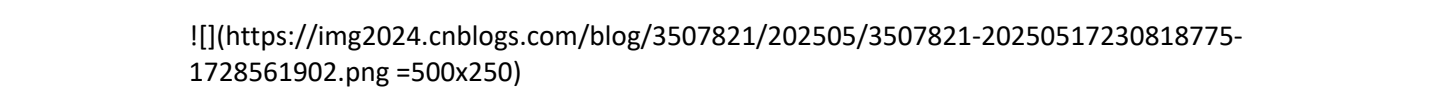
```
import java.util.concurrent.Semaphore;
  // An inner class for account
  private static class Account {
    // Create a semaphore
    private static Semaphore semaphore = new Semaphore(1);
    private int balance = 0;
    public int getBalance() {return balance;}

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

```
        }
    }
  }
}
```

死锁

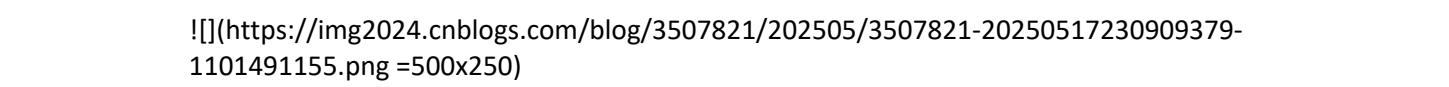
死锁：多个线种互相等待对方持有的锁，而在得到对方的锁之前都不会释放自己的锁



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

30.6 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}



30.7 内部类

30.7.1 概念和分类

- 内部类也称为嵌套类，是在一个类的内部定义的类。通常一个内部类仅被其外部类使用时,同时也不想暴露出去，才定义为内部类。JDK16以前，内部类不能定义在方法中。但是JDK16以后方法里也可以定义类 (称为方法内部类，本课程不做介绍，不作要求掌握)
    - 内部类分为实例内部类和静态内部类
- **实例内部类**内部不允许定义静态成员（JDK16以前）。从JDK16开始，实例内部类可以定义静态成员了。创建实例内部类的对象时需要使用 **外部类的实例变量.new 实例内部类类名(**)。(即只有当有了外部类的实例，才能实例化 实例内部类的对象）
- **静态内部类**用static定义，其内部允许定义实例成员和静态成员。
- 静态内部类的方法**不能访问外部类的实例成员变量**。
- 创建静态内部类的对象时需要使用new 外部类.静态内部类( )

例:

```
class Wrapper{
    private int x=0;
```

```
    private static int z = 0;
    //内部静态类
    static class A{
        int y=0;
        static int q=0;  //可以定义静态成员,
        //不能访问外部类的实例成员x，可访问外部类静态成员z
        int g() {  return ++q + ++y + ++z; }
    }
    //内部实例类,也定义静态成员（JDK16以后）
    //内部实例类可访问外部类的静态成员如z，实例成员如x
    class B{
        int y=0;
        public int g( ) {
            x++; y++;z++;
            return x+y;
        }
        public int getX(){return x;}
        //从JDK16开始，内部实例类可以定义静态成员
        static void f(){}
    }
}

public static void main(String[] args){ //和class Wrapper同一个JAVA文件，即同一个包
        Wrapper w = new Wrapper();  //w.x = 0;
        //创建内部静态类实例
        Wrapper.A a = new Wrapper.A(); //a.y=0, a.q=0;
        Wrapper.A b = new Wrapper.A();  //b.y=0, b.q=0;
        a.g();
        //a,b的实例成员彼此无关，因此执行完a.g()后，a.y = 1, b.y = 0;
        //a,b共享静态成员q，所以a.q=b.q = 1;

        //创建内部实例类实例
        //不能用new Wrapper.B();必须通过外部类对象去实例化内部类对象
        Wrapper.B  c = w.new B(); //类型声明还是外部类.内部类
        c.y=0;
        c.g(); //c.y = 1 ,c.gextX() = 1

        //在外部类体外面，不能通过内部类对象访问外部类成员，只能在内部类里面访问，
        //编译器在这里只能看到内部类成员
       //   System.out.println(a.z); //错误
       //   System.out.println(c.x); //错误
        //不能通过c直接访问外部类的x，可通过c.gextX()
        System.out.println(c.getX());
}
```

> 内部类可以被成员访问控制符修饰（私有、缺省、保护、公有的），访问控制规则和类成员访问控制一样

内部类作用：如果一个类A仅仅被某一个类B使用，且A无需暴露出去，可以把A作为B的内部类实现，

内部类也可以避免名字冲突：因为外部类多了一层名字空间的限定。例如类Wrapper1、Wrapper2可以定义同名的内部类MessageHandlerImpl而不会导致冲突

```
public class Wrapper1 {
    //定义内部类MessageHandlerImpl实现Message接口
    class MessageHandlerImpl implements MessageHandler {
        @Override
        public void handle(String message) { System.out.println(message);}
    }

    //Wrapper1的实例变量
    private MessageHandler handler = new MessageHandlerImpl();
    public void sendMessage(String message){
        handler.handle(message);
    }
    public static void main(String[] args){
        new Wrapper1().sendMessage("Message from wrapper1");
    }
}
public class Wrapper2 {
    //定义内部类MessageHandlerImpl实现Message接口
    class MessageHandlerImpl implements MessageHandler {
        @Override
        public void handle(String message) { System.out.println(message);}
    }

    //Wrapper2的实例变量
    private MessageHandler handler = new MessageHandlerImpl();
    public void sendMessage(String message){
        handler.handle(message);
    }
    public static void main(String[] args){
        new Wrapper2().sendMessage("Message from wrapper2");
    }
}
```

30.7.2 匿名内部类

匿名内部类可以简化编程。简化时使用匿名内部类的父类或者接口代替匿名内部类。
比如

```
public class Wrapper3 {
   //Wrapper1的实例变量
    private MessageHandler handler = null;
    public void setHandler(MessageHandler handler){ this.handler = handler;}
    //定义内部类MessageHandlerImpl实现Message接口
    class MessageHandlerImpl implements MessageHandler {
        @Override
        public void handle(String message){ System.out.println(message);}
    }
    public void init(){
        setHandler(new MessageHandlerImpl());
    }
}
```

MessageHandlerImpl这个类名其实不重要，重要的是需要实现MessageHandler接口，因此想去掉类名，这就是匿名内部类。但new
后面必须有一个类型名，就用这个类所实现的接口名作为匿名内部类的类名。

```
public class Wrapper3 {
    //其它代码省略
    public void init(){
        setHandler(new MessageHandler(){
            @Override
            public void handle(String message) { System.out.println(message);}
        });
    }
}
```

- 匿名内部类可以简化编程。简化时使用匿名内部类的**父类或者所实现接口**代替匿名内部类名字，作为new后面的类型。
- 匿名内部类总是使用**父类**的无参构造方法产生实例，对于接口使用Object（ ）。
- 匿名内部类必须实现**父类或者接口**的所有抽象方法。事件处理接口通常只有1个方法。
- 一个匿名内部类被编译成OuterClassName$n.class,如Test$1.class, Test$2.class

30.8 Lambda表达式

30.8.1 概念

Lambda表达式可以进一步简化事件处理的程序编写
编译器会将lamda表达式看待为**匿名内部类对象**，将这个对象理解为实现了MessageHandler接口的实例。下面例子中因为MessageHandler接口定义了参数为String类型的方法handle，因此编译器可以推断参数message的类型为String，并且message->{ }中右边的{ }就是handle方法的方法体。
MessageHandler接口只有一个方法，只有一个方法的接口称为功能接口（**函数式接口**），每个**Lambda 表达式都能隐式地赋值给函数式接口**，lamda表达式中的{
}就是函数式接口中接口方法的方法体。

```
setHandler(message ->{
      System.out.println(message);
});
```

Lambda表达式本质上更像匿名函数。
Java里规定Lambda表达式只能赋值给函数式接口。
Lambda表达式的语法为：
（type1 para1, …, typen paran)->expression 或者
（type1 para1, …, typen paran)->{ 一条或多条语句}
当把Lambda表达式赋值给函数式接口时，
Lambda表达式的参数的类型是可以推断的；**如果只有一个参数，则可以省略圆括弧**。从而使Lambda表达式简化为：
e->处理e的expression 或者
e->{ 处理e的statements; }

30.8.2 Lambda 表达式的结构

一个 Lambda 表达式可以有零个或多个参数
参数的类型既可以明确声明，**也可以根据上下文来推断**。例如：(int a)与(a)效果相同（当可以推断类型时）
所有参数需包含在圆括号内，参数之间用逗号相隔。例如：(a, b) 或 (int a, int b) 或 (String a, int b, float c)
空圆括号代表参数集为空。例如：() -> 42
当**只有一个参数**，且其**类型可推导**时，**圆括号（）可省略**。例如：a -> {return a*a；}
Lambda
表达式的主体可以是表达式或者是block，如果是**表达式，不能有{}**；如果是block，则必须加 { }

每个 Lambda 表达式都能隐式地赋值给函数式接口
Runnable接口就是函数式接口，里面定义接口方法void run( )，我们可以通过 Lambda 表达式创建一个接口实例 。

```
Runnable   r = ( ) -> System.out.println("hello world");
```

上面语句的含义是：将一个实现了Runnable接口的类的实例赋值给Runnable接口引用r， Lambda 表达式的主体就是接口方法void run( )的具体实现
当不是显式赋值给函数式接口时，编译器会自动解释这种转化：

```
new Thread(
       () -> System.out.println("hello world")
).start();
```

在上面的代码中，编译器会自动推断：根据线程类的构造函数签名 public Thread(Runnable r) { }，将该 Lambda 表达式赋给 Runnable 接口。

30.8.3 Lambda表达式用法实例

函数式接口定义好后，我们可以在 API 中使用它，同时利用 Lambda 表达式。

```
//定义一个函数式接口
public interface WorkerInterface {
       public void doSomeWork();
}

public class WorkerInterfaceTest {
       public static void exec(WorkerInterface worker) {
              worker.doSomeWork();
       }

       public static void main(String [] args) {
              //invoke doSomeWork using Annonymous class
              exec( new WorkerInterface() {
                     @Override public void doSomeWork() {
                     System.out.println("Worker invoked using Anonymous class"); }
              } );
              //invoke doSomeWork using Lambda expression
              exec( () -> System.out.println("Worker invoked using Lambda expression")
);
       }
 }
```

例:

```
public class LambdaDemo {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(2);
        //传入匿名内部类，Runnable接口实例
        executor.execute(new Runnable() {
            @Override
            public void run() { System.out.println("Runnable 1"); }
        });
        //传入Lambda表达式，Runnable接口实例，右边是Statements，必须放在{}
        executor.execute(()->{System.out.println("Runnable 2");});
        //传入Lambda表达式，Runnable接口实例，右边是expression，不能放在{}里，不带;
        executor.execute(()->System.out.println("Runnable 3"));

        executor.shutdown();
    }
}
```

Lambda 神奇功能

计算给定数组中每个元素平方后的总和。请注意，Lambda
表达式只用一条语句就能达到此功能，这也是 MapReduce 的一个初级例子。我们使用 map()
给每个元素求平方，再使用 reduce() 将所有元素计入一个数值：
java.util.stream.Stream 接口包含许多有用的方法，能结合 Lambda 表达式产生神奇的效果。

```
//Old way:
List<Integer> list = Arrays.asList(1,2,3,4,5,6,7);
int sum = 0;
for(Integer n : list) {
       int x = n * n;
       sum = sum + x;
 }
System.out.println(sum);

//New way:
List<Integer> list = Arrays.asList(1,2,3,4,5,6,7);
 int sum = list.stream().map(x -> x*x).reduce((x,y) -> x + y).get();
System.out.println(sum);
```

30.8.4 Java Lambda 表达式的由来

Java
中的一切都是对象（除了基本数据类型），即使数组也是一种对象，每个类创建的实例也是对象。在 Java
中定义的函数或方法不可能完全独立，**不能将方法作为参数或返回一个方法**。为此，Java 8
增加了一个语言级的新特性，名为 Lambda 表达式。
在函数式编程语言中，函数是一等公民，它们可以独立存在，你可以将其赋值给一个变量，或将他们当做参数传给其他函数。JavaScript
是最典型的函数式编程语言（当然也是面向对象的）。函数式语言提供了一种强大的功能——闭包。当一种编程语言支持函数返回类型为函数时，这种语言天然就支持闭包。
Java虽然不支持函数返回类型为函数，但可以用匿名内部类实现闭包，但这种闭包多了一个限制：要求捕获的的自由变量必须是final的。用Lambda表达式同样如此：

Lambda表达式捕获的自由变量必须是final的
为什么Java里的闭包多了这个限制：在Java的经典著作《Effective Java》、《Java Concurrency in Practice》大神们这么解释：如果Java闭包捕获的自由变量是非final的，会导致线程安全问题。Python和Javascript则不用考虑这样的问题，所以它的闭包捕获的自由变量是可以任意修改的。