

# JAVA程序员培训-1

讲师:陈伟俊

# 研究课题

- Java入门
- 面向对象的特征
- 图形用户界面
- 多线程
- 数据I/O
- 网络编程

# 第一章Java入门

- JAVA编程语言基础
- 面向对象的程序设计
- 标识符关键字数据类型
- 运算符表达式和流程控制

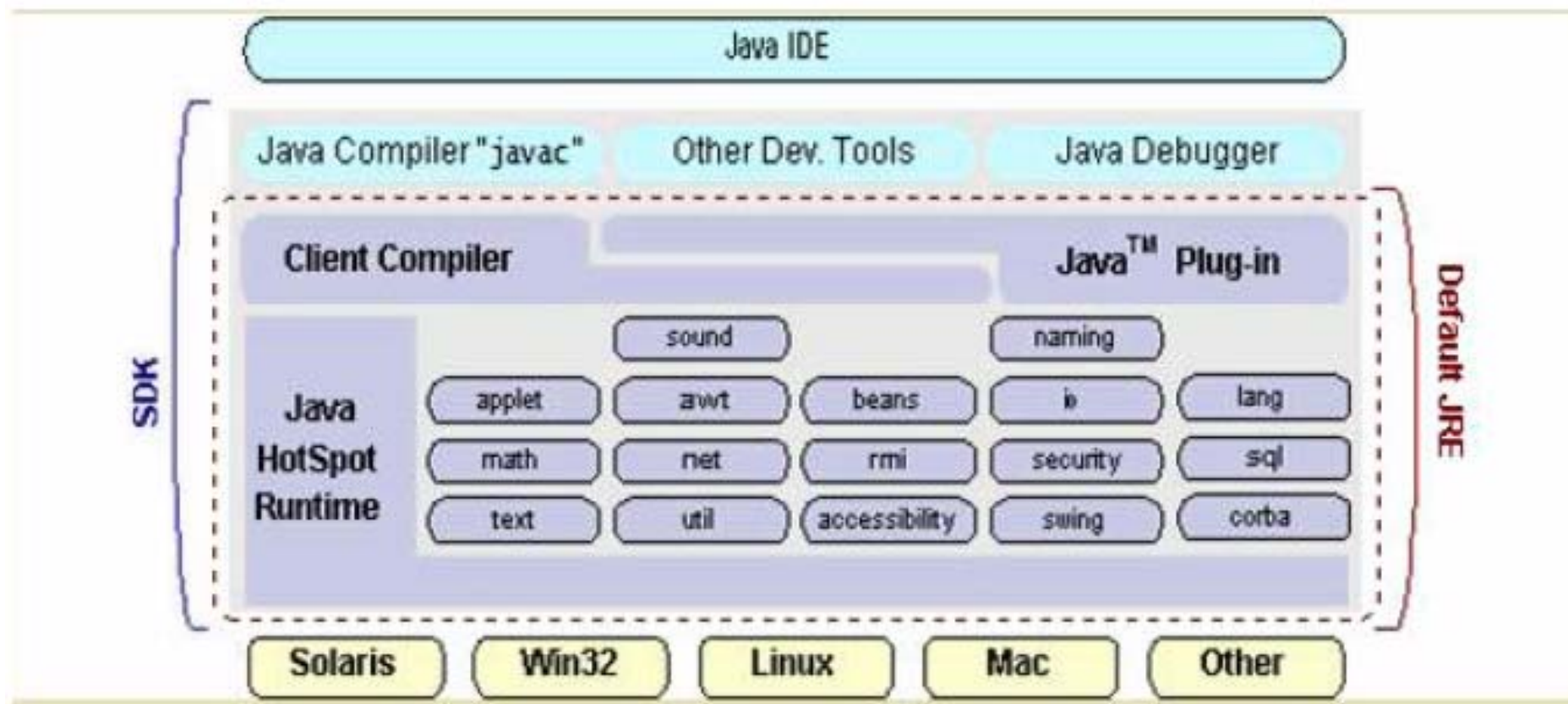
# JAVA编程语言基础

# JAVA概述

什么是java?

- 是一种编程语言
- 是一种开发环境
- 是一种应用程序环境
- 是一种部署环境

# JAVA概述



# Java技术的主要目标

- 提供一种易于编程的语言
- 提供一种解释环境（JVM）
- 用户能够运行多个活动线程
- Java支持程序运行期间动态修改代码模块
- 提供代码验证机制以保证安全性

# Java的三种核心机制

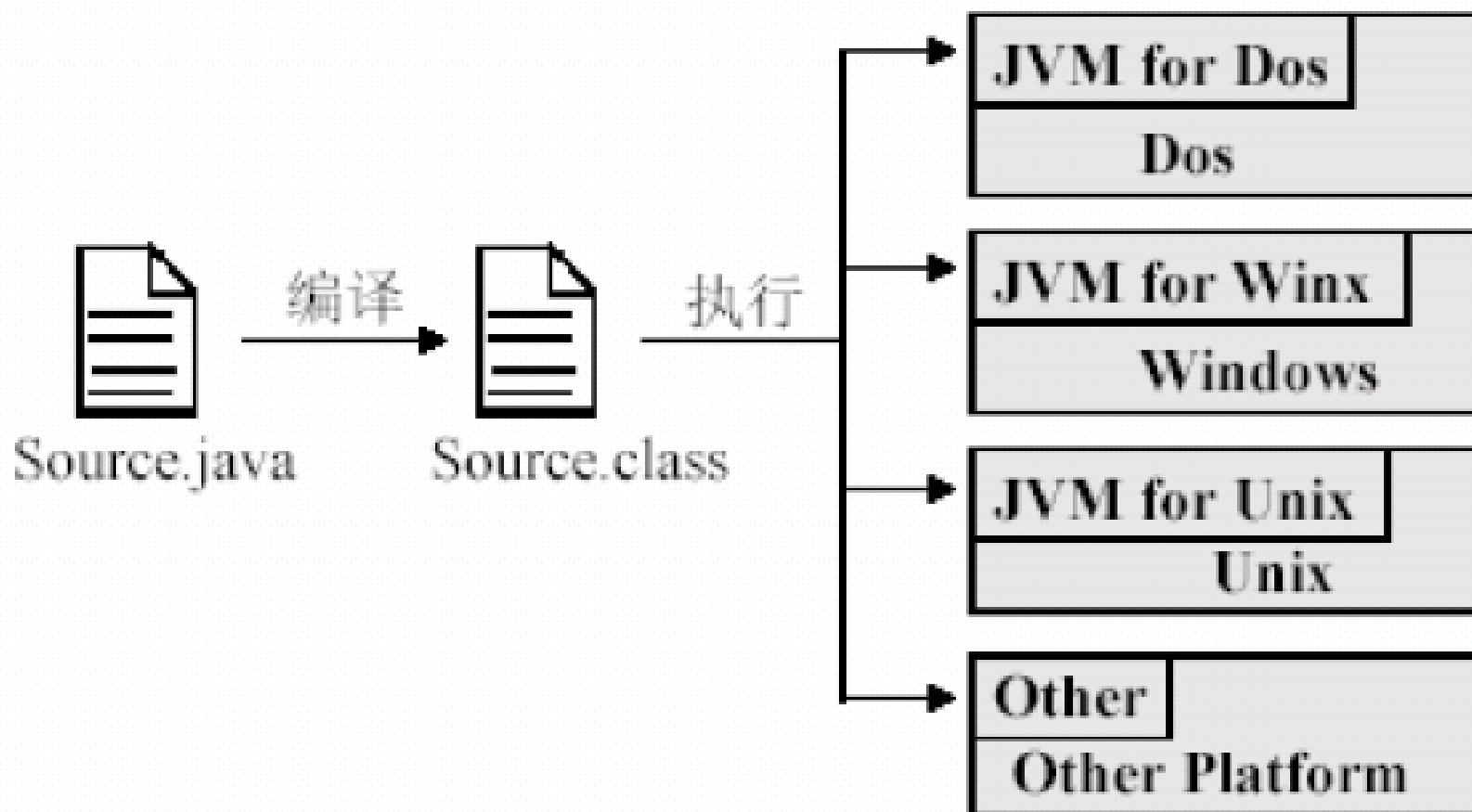
- Java虚拟机(Java Virtual Machine)
- 垃圾收集机制(Garbage collection)
- 代码安全性检测(Code Security)



# Java虚拟机（JVM）

- 提供硬件平台规范
- 读取独立于平台的字节码
- 在JAVA开发工具或浏览器中都能实现

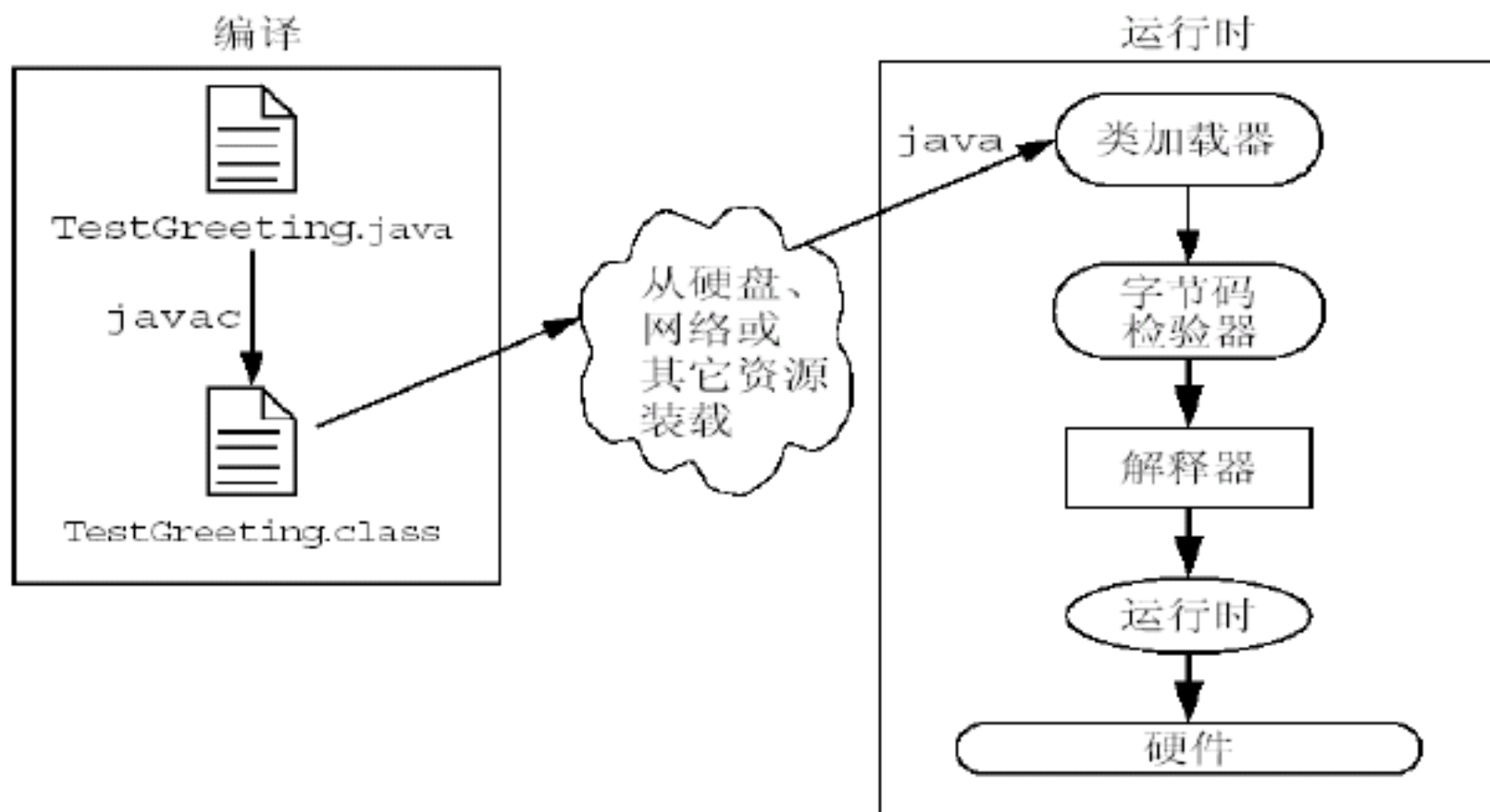
# Java虚拟机 (JVM)



# 垃圾收集

- 已分配的内存不需要的时候应释放
- 垃圾收集在Java程序运行过程中自动进行  
它提供一种系统级线程跟踪存储空间的分配情况。并在JVM的空闲时，检查并释放那些可被释放的存储器空间。  
程序员无法精确控制和干预。在JVM的实现中，可能会有很大差异。

# 代码安全性



# 代码安全性

Java运行时环境

- 加载代码
- 校验代码
- 执行代码

# 代码安全性

## 类加载器

- 加载程序执行需要的全部类
- 使用“命名空间”保持本地文件系统的类分离
- 防止欺骗

# 代码安全性

字节码校验器

保证：

- 类符合JVM规范指定的类文件格式
- 没有访问限制违例
- 代码未引起操作数堆栈上溢或下溢
- 所有操作代码的参数类型是正确的
- 无非法数据转换发生

# 开发工具包

JDK---Java Develop Kit

J2SDK---Java 2(platform) Software Developing Kit

三种版本

- 标准版 (Standard Edition) 又称j2se
- 企业版 (Enterprise Edition) 又称j2ee
- 微型版 (Micro Edition) 又称j2me



# 配置环境变量

Path 外部命令搜索路径

Classpath 类资源位置搜索路径

- 临时配置：仅使用一次
- 持久配置：一直使用
  - 用户变量：对当前用户有效
  - 系统变量：对所有用户有效（需重启机器）

# 编辑代码工具

- 记事本
- EDITPLUS
- ULTRAEDIT
- ECLIPSE
- JBUILDER
- .....

# 编写第一个JAVA程序

Java应用程序（application） HelloWorld.java

注意项

- 类名和文件名相同
- main方法必须按照规则，不能有任何的改变
- 注意区分大小写
- 以；结束语句
- 扩展名.java

# 编写第一个JAVA程序

- 应用程序的入口
  - main() 方法
    - 应用程序的入口是main() 方法，有固定的书写格式
    - `public static void main(String args[]) {`
    - ...
    - }
- 应用程序的编译
  - `javac类名.java`
  - 例如: `javacHelloWorld.java`
- 产生类文件(.class)
  - 例如: 产生HelloWorld.class
- 应用程序的运行
  - `java 类名`
  - `java HelloWorld`

# 本章结束

## 本章回顾

# 第二章 面向对象的编程

面向对象思想

➤万物皆为对象

面向对象程序设计

➤将现实世界中的事物用计算机编程语言描述

# 类和对象

类 (class)

- 对象的蓝图，生成对象的模板，是对一类事物的描述，是抽象的概念上的定义

对象 (object)

- 对象是实际存在的该类事物的每个个体。因而也成为实例 (instance)

# 类的面向编程三大特性

- 封装
- 继承
- 多态



# 类和对象举例

## 定义类

```
public class Animal {  
    public int legs;  
    public void eat() {  
        System.out.println("eat");  
    }  
}
```

Animal
<b>+legs:int</b>
<b>+eat()</b>

## 生成类的对象实例

```
Animal myCat = new Animal()
```

# 声明类

语法格式：

```
[< 修饰符>] class < 类名> {  
    [<属性声明>]  
    [<构造器声明>]  
    [<方法声明>]  
}
```

# 声明属性

语法格式：

[<修饰符>] 类型< 属性名> [=初值] ;

例：

```
public int legs;
```

```
public int age = 20;
```

# 声明方法

语法格式：

```
[<修饰符>] <返回类型> <方法名>([<参数表>]) {  
    [< 语句>]  
}
```

例：

```
public void eat() {  
    System.out.println("eat");  
}
```

# 练习

- 编写一个计算器Computer类，实现两个方法：返回两个数的相加值和两个数的相减值
- 编写一个TestComputer类，测试打印Computer类的两个方法的返回值是否正确

# 对象成员的访问

➤“点”符号：〈对象〉.〈成员〉

➤成员包括属性，方法

➤举例：

```
Animal xb= new Animal();
```

```
xb.legs=4;
```

```
xb.eat();
```

```
xb.move();
```

```
xb.setLegs(2);
```

```
int num = xb.getLegs();
```

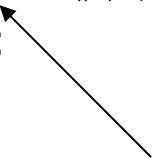
# 练习

- 写一个Cat类，定义属性weight（重量），定义方法showWeight能够返回weight的值。
- 写一个测试类TestCat，设置weight属性值为10，调用showWeight方法输出属性值

Cat
+weight:int
+showWeight(): int

# 在方法中调用其他方法

```
public class TestCallMethod{  
    public void m1() {  
        System.out.println("m" + m2());  
        System.out.println("m1");  
    }  
  
    public int m2() {  
        return 2;  
    }  
  
    public static void main(String args[]) {  
        TestCallMethod tcm = new TestCallMethod();  
        tcm.m1();  
    }  
}
```



此处产生断点，转到m2方法体



# 在方法中调用其他方法

```
• public class TestCallMethod{  
•     public void m1() {  
•         m2(); ← 产生断点，转到m2方法体  
•         System.out.println("m1");  
•     }  
•     public void m2() {  
•         m3(); ← 产生断点，转到m3方法体  
•         System.out.println("m2");  
•     }  
•     public void m3() {  
•         System.out.println("m3");  
•     }  
•     public static void main(String args[]) {  
•         TestCallMethod tcm = new TestCallMethod();  
•         tcm.m1();  
•     }  
• }
```

# 方法的递归调用

- 在方法体中调用本方法
  - 不加判断会形成无限调用
- 例：求某个数的阶乘值

```
public class TestLoopCall{
    public int getResult(int i){
        if (i == 1) {
            return 1;
        }
        return i * getResult(i - 1); //调用本方法
    }
    public static void main(String args[]){
        TestLoopCall tlc = new TestLoopCall();
        System.out.println(tlc.getResult(5)); // 1*2*3*4*5的结果是120
    }
}
```

# 信息的封装引例

```
public class Animal {  
    public int legs;  
    public void eat() {  
        System.out.println("eat");  
    }  
}
```

```
public class TestAnimal {  
    public static void main(String args[]) {  
        Animal myCat = new Animal();  
        myCat.legs = -1000;    //属性赋值与实际不符合!  
    }  
}
```

# 信息的隐藏和封装

## ➤ 问题

- 用户代码直接访问内部数据（类定义的属性）会导致数据的错误、混乱和安全性差等问题

## ➤ 解决

- 将类属性的访问私有化（private），不允许用户代码直接访问
- 用户只能通过公有方法（public）对属性进行存取

# 信息的隐藏和封装实例

```
public class Animal{
    private int legs;           //将属性legs定义为private，只能被该类的成员访问
    public void setLegs(int i){
        if (i != 0 && i != 2 && i != 4){
            System.out.println("Wrong number of legs!");
            return;
        }
        legs = i;
    }
    public int getLegs(){
        return legs;
    }
}

public class Zoo{
    public static void main(String args[]) {
        Animal xb=new Animal();
        xb.setLegs(4);          //xb.setLegs(-1000);          xb.legs=-1000;//非法
        System.out.println(xb.getLegs());
    }
}
```

# 信息的隐藏和封装总结

- 将类属性声明为私有（private），提供公有的（public）方法setXXX和getXXX对数据进行存取，实现下列目标：
  - 隐藏一个类的实现细节；
  - 使用者只能通过事先定制好的方法来访问数据，可以方便地加入控制逻辑，限制对属性的不合理操作；
  - 便于修改，增强代码的可维护性；

# 练习1

编写一个卡车类，设置一个load属性表示载重量，提供get和set方法，以及增加载重量的addLoad方法，load最大不能超过500

Truck
-load:int
+setLoad(l:int) +getLoad():int +addLoad(l:int)

## 练习2

编写Birthday类，如右图, 封装year和month属性，自己编写setXXX和getXXX方法，提供show方法可以打印生日

格式（年/月）

Birthday
-year:int -month:int
..... +show()



# 构造器

## ➤ 语法格式:

```
[<修饰符>] <类名>([< 参数表>]) {  
    [< 语句>]  
}
```

## ➤ 例:

```
public class Cat {  
    private int weight;  
    public Cat() {  
        weight = 5;  
    }  
    .....  
}
```

# 默认构造器

- 在java中，每个类都必须有至少一个构造器
- 如果类中没有显式的定义任何构造方法，系统会自动的提供一个默认构造方法
  - 默认构造器没有参数
  - 默认构造器方法体为空

```
public class Cat {  
    private int weight;  
    public Cat() { //系统提供  
    }
```

```
.....  
}
```

# 一个java源文件中

## ➤ 包含

[包声明]

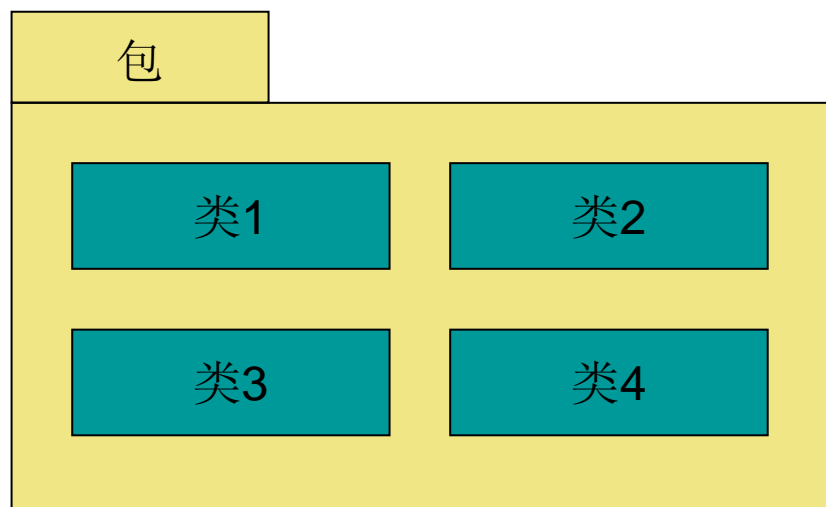
[导入声明]+

[类声明]+

# 包(package)

## ➤ 作用

- 区分名称相同的类
- 能有效实施访问权限控制
- 有助于软件功能分类



# package语句

- 位置：
  - 作为源文件的第一行有效代码指明该文件中定义的类所在的包。
- 语法格式：
  - `package <顶层包名>[.<子包名>]* ;`
- 提示：
  - 包的目录层次用“.”间隔
  - 包通常用小写单词
- 举例
  - `package com.dhee.test;`

# import语句

## ➤ 作用

- 将不同包中的类导入

## ➤ 语法格式

- `import 包名[. 子包名...]. <类名|*>`

## ➤ 注意

- 导入语句仅导入当前包中所有类，而不包括子包

## ➤ 举例

- `import com.dhee.test.*;`

# java中提供的基本包

- `java.lang`----包含一些Java语言的核心类，如 `String`、`Math`、`Integer`、`System`和`Thread`，提供常用功能。
- `java.awt`----包含了构成抽象窗口工具集（abstract window toolkits）的多个类，这些类被用来构建和管理应用程序的图形用户界面(GUI）。
- `java.net`----包含执行与网络相关的操作的类。
- `java.io`----包含能提供多种输入/输出功能的类。
- `java.util`----包含一些实用工具类，如定义系统特性、使用与日期日历相关的函数。
- `java.lang`包系统会自动导入，无须显式的`import`导入

本章结束



# 第三章 标识符、关键字和数据类型

- 在java中允许有三种类型的注释
  - `//` 单行注释          注释到行尾
  - `/*` 单行或多行注释`*/`
  - `/**` 可以用于文档化处理的单行或多行注释`*/`
- 注意
  - `/*` `*/`不能嵌套使用

# 点、分号、空白、花括号

## ➤ 各符号作用

- 使用对象的成员、包名和子包名用点分隔
- 一条java语句用；结束
- 在java程序中可以有任意数量的空白（包括空格和回车换行）
- 一条或多条语句可以用花括号{}括起来形成一个语句块，如类体、方法体、循环体等，语句块可以相互嵌套层数无限制。

# 标识符

- 是指给程序中的包、类、接口、变量或方法起的名字。
- 命名规则
  - 应以字母、下划线、美元符开头
  - 后跟字母、下划线、美元符或数字
  - 不能用关键字和保留字
  - 严格区分大小写
  - 名称长度没有限制

# 标识符举例

合法标识符	非法标识符
helloWorld	Hello word
class1	class
arr_cat	arr-cat
_4587	458.7
\$9	9\$

# 关键字

- 在java中一些特殊的单词，在程序中有专门的用途，称为关键字，所有关键字均为小写
- 特别说明
  - const和goto不是关键字，它们是保留字
  - true、false、null虽被用做专门用途，但不是Java关键字
  - Java5以后关键字和保留字也概称关键字

# 关键字列表

abstract	default	if	private	this
boolean	do	implements	protected	throw
break	double	import	public	throws
byte	else	instanceof	return	transient
case	extends	int	short	try
catch	final	interface	static	void
char	finally	long	strictfp	volatile
class	float	native	super	while
const	for	new	switch	
continue	goto	package	synchronized	

# 数据类型

➤ Java是强类型语言，每个变量都必须声明类型

# 数据类型划分





# 逻辑型

- `boolean` 类型适合逻辑条件判断，一般用于程序流程控制
- 只能取两个值 `true` 和 `false`，不能取 0 或非 0 等整数，不能和整数互相转换

# 字符型

- 字符型数据使用单引号括起来的单个字符
  - 例 `char c = 'a';`
- Java字符采用双字节（16位）的unicode字符，支持世界上所有的书面语言，可以用16进制编码表示
  - 例 `char c1 = '\u0061';` //代表字母a
- 除了表示unicode的u的转义符之外，还有用于特殊字符的转义序列
  - 例 `char c2 = '\n'` //表示换行符
- Unicode字符能否显示和操作系统有直接关系

# 整型

- Java中整型表示无小数部分的有符号数字
- Java提供了四种整型
- java整数范围与机器平台无关，保证代码移植性

类型	占用存储空间(字节)	取值范围
byte	1	$-2^7 \sim 2^7 - 1$ -128 ~ 127
short	2	$-2^{15} \sim 2^{15} - 1$ -32768 ~ 32767
int	4	$-2^{31} \sim 2^{31} - 1$ -2147483648 ~ 2147483647
long	8	$-2^{63} \sim 2^{63} - 1$ -92233720368554775808 ~ 92233720368554775807

# 整型

- Java语言整型常量的三种表示形式：
  - 十进制整数，如12, -314, 0。
  - 八进制整数，要求以0开头，如012
  - 十六进制数，要求0x或0X开头，如0x12
    - 如： `inta1=17;`
    - `inta2=017; //inta2=15;`
    - `inta3=0x17; //inta3=23;`
- Java语言的整型常量默认为int型，如：
  - `inti =3;`
- 声明long型常量可以加'l '或'L '，如：
  - `long l = 3L;`

# 浮点型

➤ 浮点类型表示有小数部分的数字

类型	占用存储空间(字节)	取值范围
float	4	-3.403E38~3.403E38
double	8	-1.798E308~1.798E308

# 浮点型

- Java浮点类型常量有两种表示形式
  - 十进制数形式，必须含有小数点，例如：
    - 3.14 314.0 .314
  - 科学记数法形式，如
    - 3.14e2 3.14E2 314E2
- Java浮点型常量默认为double型，如要声明一个常量为float型，则需在数字后面加f或F，如：
  - double d = 3.14;
  - float f = 3.14f;

# 基本数据类型变量声明和赋值

- `boolean b = true;` //声明boolean型变量并赋值
- `int x, y = 8;` // 声明int型变量
- `float f = 4.5f;` // 声明float型变量并赋值
- `double d = 3.1415;` //声明double型变量并赋值
- `char c;` //声明char型变量
- `c = '\u0031';` //为char型变量赋值
- `x = 12;` //为int型变量赋值

# Java类的成员变量默认初始化

➤ 创建对象时，类中的成员变量会被自动初始化赋予默认值

```
public class Test{  
    int i;  
    boolean t;  
    public static void main(Stringargs[]) {  
        Test t=new Test(); //创建对象时，类中的基本数据类型属性初始化为默认值  
        System.out.println(t.i); //0  
        System.out.println(t.t); //false  
    }  
}
```



# Java类的成员变量默认初始化原则

成员变量类型	默认值
byte	0
short	0
int	0
long	0L
char	'\u0000'
float	0.0F
double	0.0D
boolean	false
所有引用类型	null

# 引用类型

- 除了基本数据类型之外的类型就是引用类型
- 引用类型可以存储复杂的数据结构
- 引用类型的数据以对象的形式存在
- 例

```
Cat c1 = new Cat();
```

//创建了一个引用变量c1，并用new创建了一个实例，其中c1指向这个实例

# Java引用类型应用举例

```
public class MyDate{
    private int day = 12;
    private int month = 6;
    private int year = 1900;
    public Mydate(int d, int m, int y) {
        year = y;
        month = m;
        day = d;
    }
    public void display() {
        System.out.println(year+ " / " + month + " / " +day);
    }
    public static void main(String[] args) {
        MyDate m;
        m = new MyDate(22, 9, 2001);
        m.display();
    }
}
```

# 范例程序流程解析图1

```
MyDate m;
```

```
m = new MyDate(22, 9, 2001);
```

```
m.display();
```

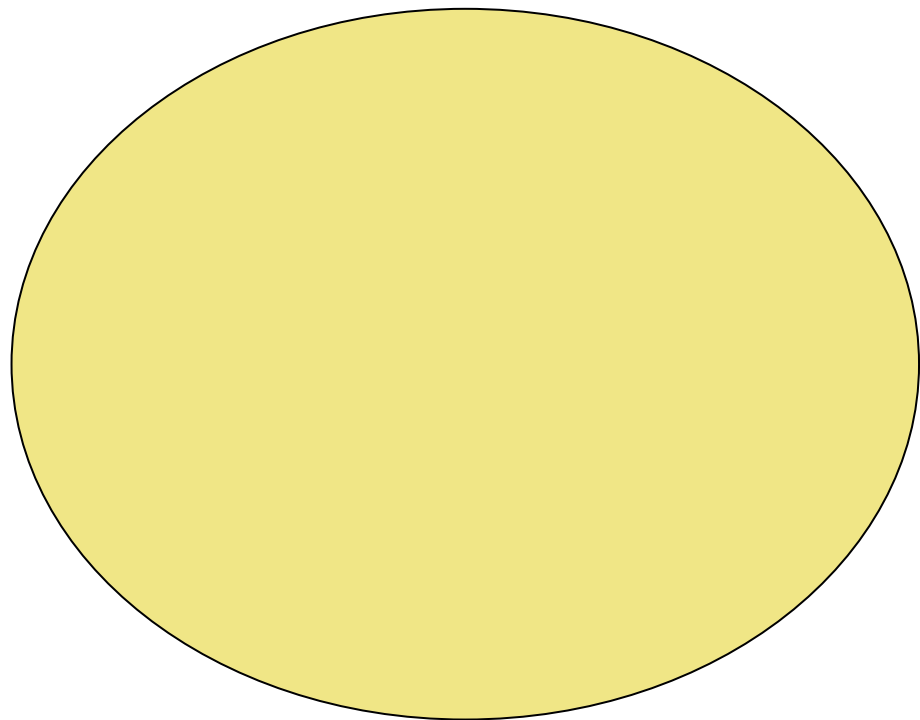
当前给MyDate引用类型的引用  
变量分配内存

栈

m

null

堆



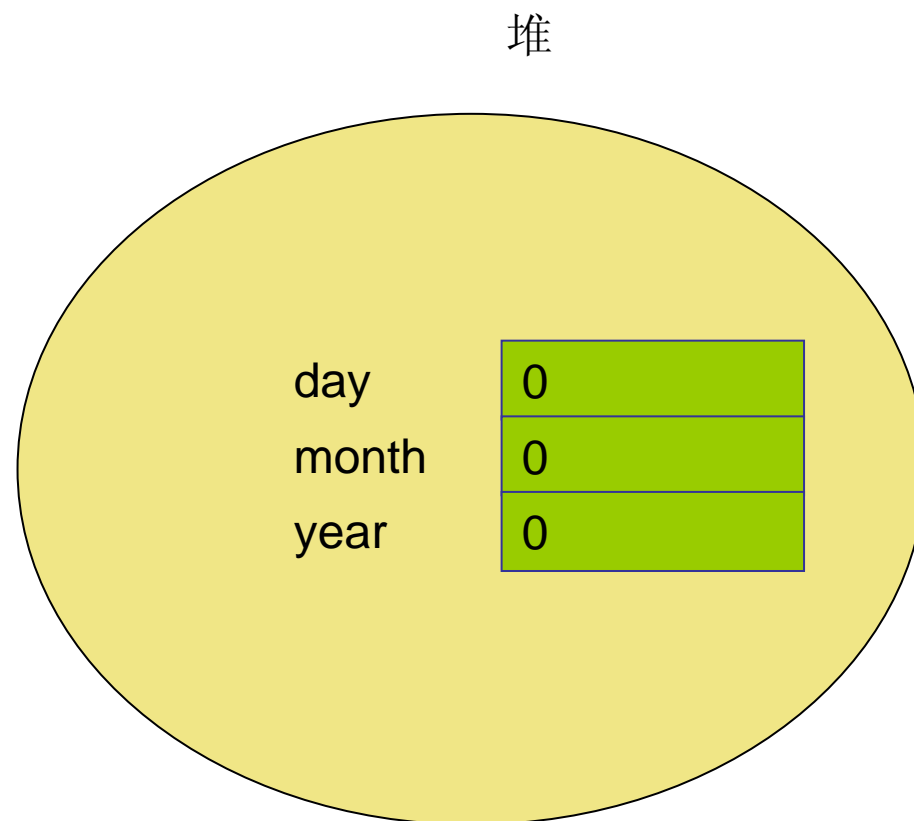
# 范例程序流程解析图2

```
MyDate m;
```

```
m = new MyDate(22, 9, 2001);
```

```
m.display();
```

创建新对象, 用默认值初始化属性值



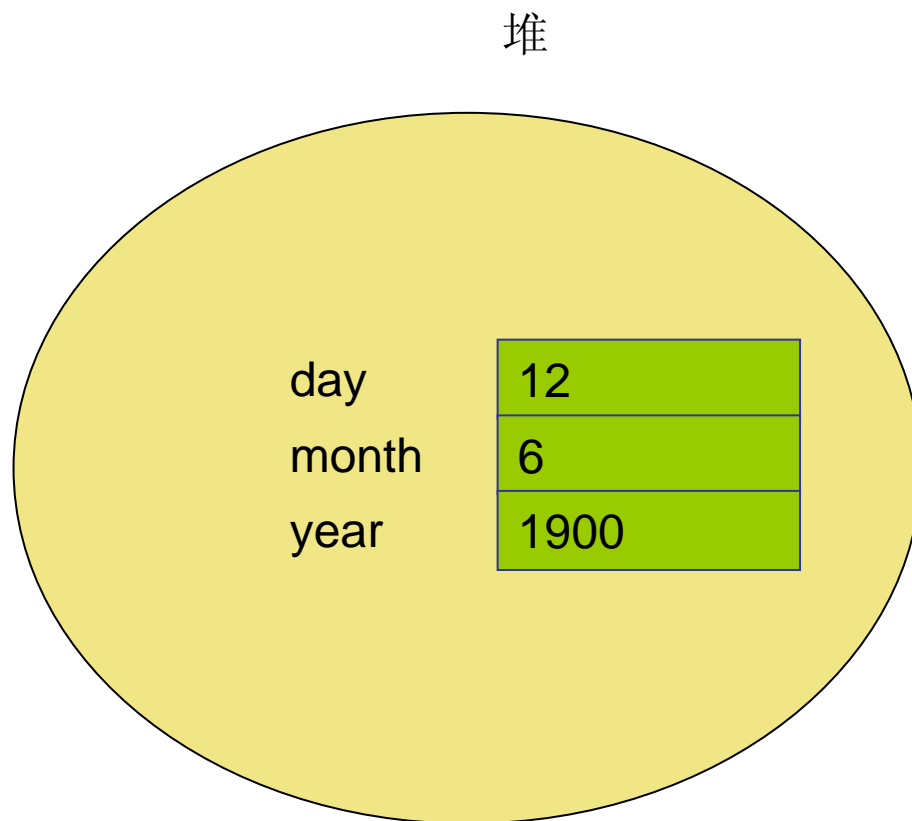
# 范例程序流程解析图3

```
MyDate m;
```

```
m = new MyDate(22, 9, 2001);
```

```
m.display();
```

在属性声明中进行了显式赋值



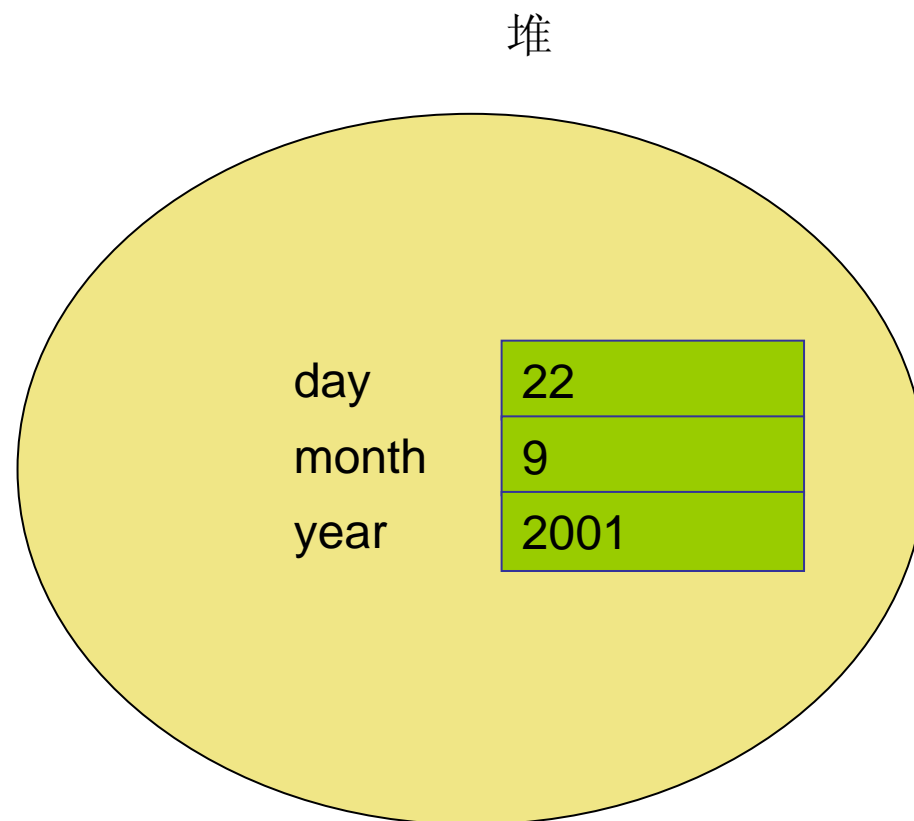
# 范例程序流程解析图4

```
MyDate m;
```

```
m = new MyDate(22, 9, 2001);
```

```
m.display();
```

在构造方法中进行了赋值



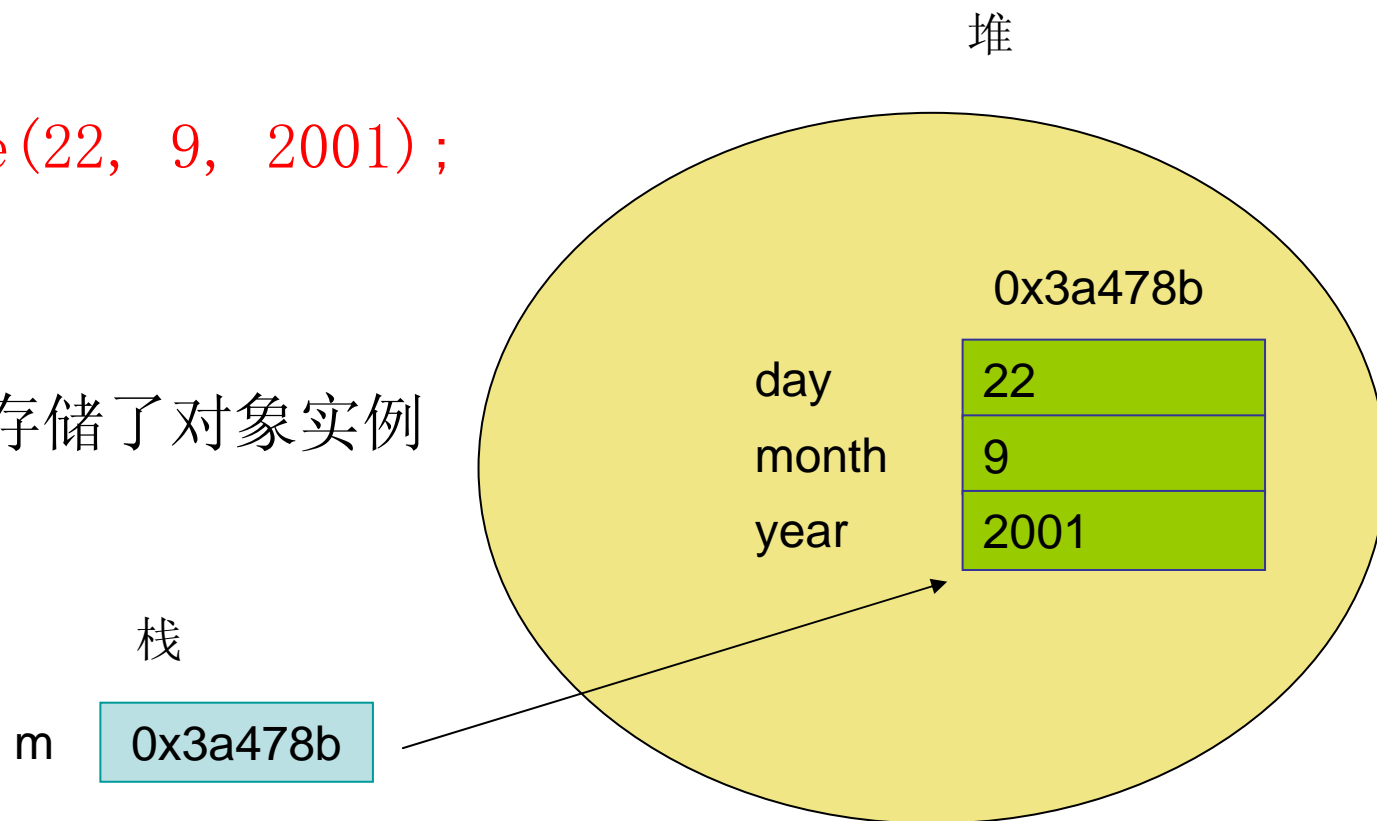
# 范例程序流程解析图5

```
MyDate m;
```

```
m = new MyDate(22, 9, 2001);
```

```
m.display();
```

在引用变量m中存储了对象实例的地址

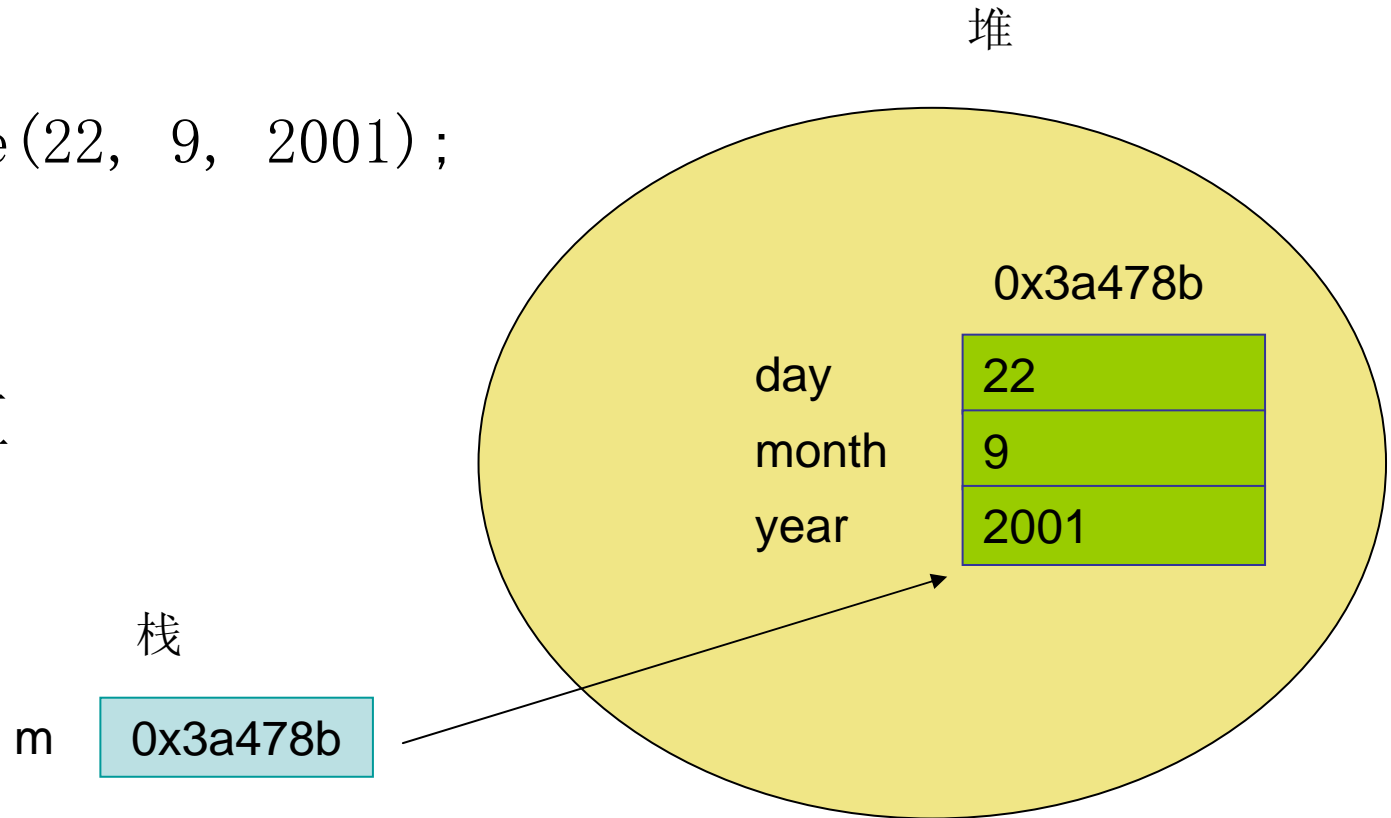




# 范例程序流程解析图6

```
MyDate m;  
m = new MyDate(22, 9, 2001);  
m.display();
```

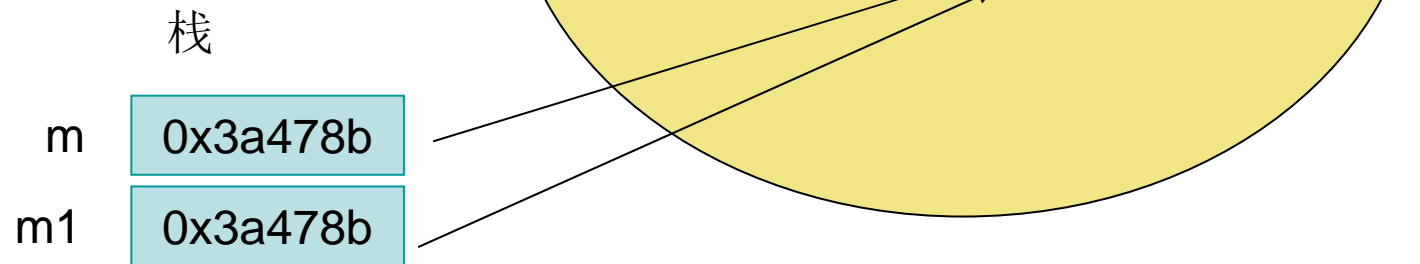
打印最后赋的值  
2001 / 9 / 22



# 引用的赋值图1

```
MyDate m;  
m = new MyDate(22, 9, 2001);  
MyDate m1 = m; //假如
```

将m1赋了一个引用变量，结果  
两个引用变量指向了同一个  
实例



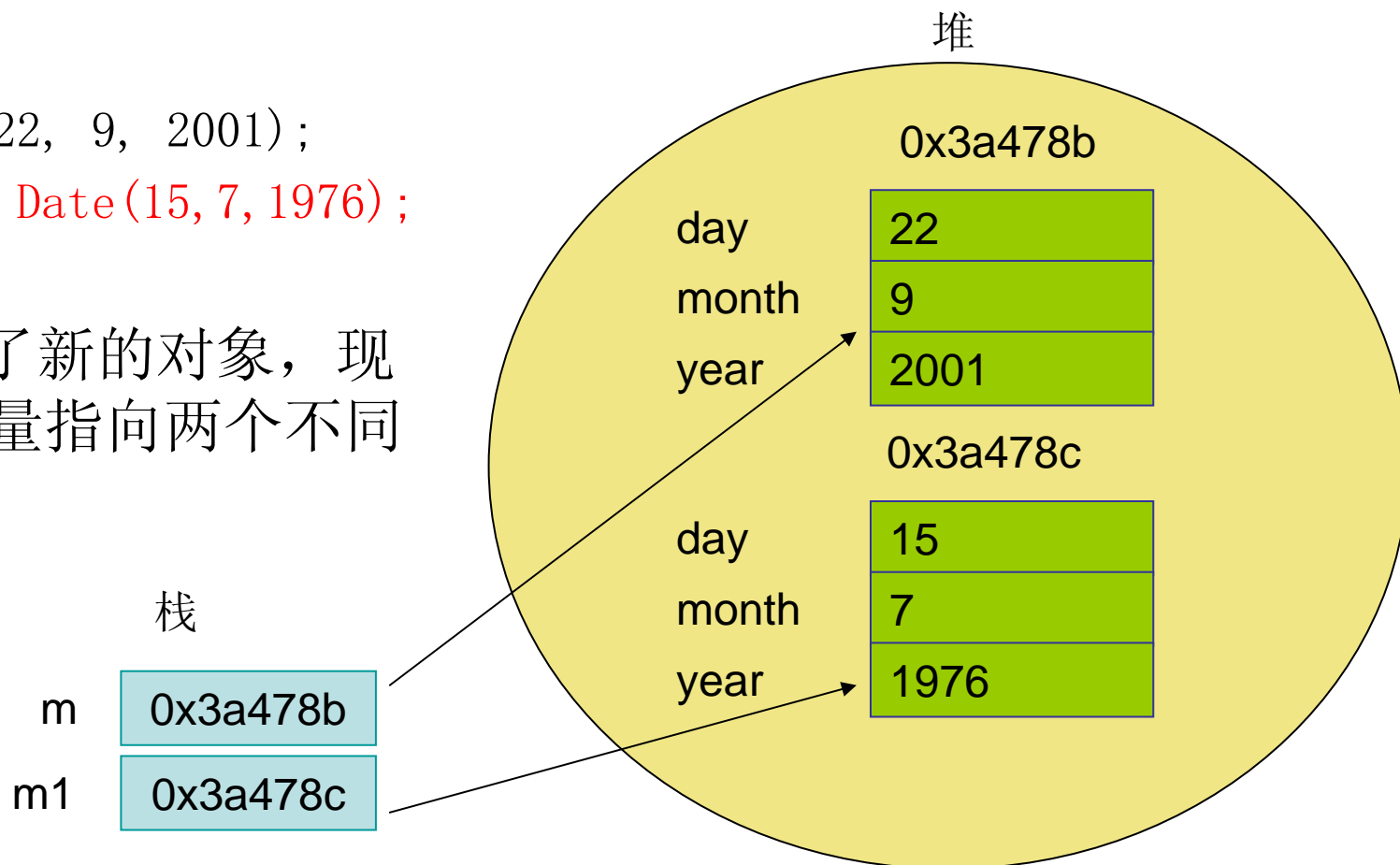
# 引用的赋值图2

```
MyDate m;
```

```
m = new MyDate(22, 9, 2001);
```

```
MyDate m1 = new Date(15, 7, 1976);
```

假如m1又生成了新的对象，现在  
是两个变量指向两个不同的  
对象实例



# 值传递

- Java中进行赋值操作或函数调用中传递参数时，遵循值传递的原则：
  - 基本类型数据传递的是该数据的值本身
  - 引用类型数据传递的是对对象的引用（句柄），而非对象本身。

# 值传递举例

```
class BirthDate{
    private int day;
    private int month;
    private int year;
    public BirthDate(int d,int m,int y){
        day = d; month = m; year = y;
    }
    public void setDay(int d) { day = d; }
    public void setMonth(int m) { month = m; }
    public void setYear(int y) { year = y; }
    public int getDay() { return day; }
    public int getMonth() { return month; }
    public int getYear() { return year;}
    public void display(){
        System.out.println(day+ " -" + month + " -" + year);
    }
}
```

# 值传递举例

```
public class Example{
    public static void main(String args[]) {
        Example ex = new Example();
        int date = 9;
        BirthDate d1= new BirthDate(7, 7, 1970);
        BirthDate d2= new BirthDate(1, 1, 2000);
        ex.change1(date);
        ex.change2(d1);
        ex.change3(d2);
        System.out.println("date=" + date);
        d1.display();
        d2.display();
    }
    public void change1(int i) {i = 1234;}
    public void change2(BirthDate b) {b = new BirthDate(22, 2, 2004);}
    public void change3(BirthDate b) {b.setDay(22);}
}
```

# 关键字this

- Java中为解决变量的命名冲突和不确定性问题，引入关键字“this”代表其所在方法的当前对象。
  - 构造方法中指该方法所创建的新对象
  - 普通方法中指调用该方法的对象

# Java 编码惯例

## ➤ 命名惯例:

➤ 包名 `package banking.domain;`

➤ 类名 `class SavingsAccount`

➤ 接口名 `interface Account`

➤ 方法名 `balanceAccount()`

➤ 变量名 `currentCustomer`

➤ 常量名 `HEAD_COUNT`



本章结束

# 第四章 运算符、表达式和流程控制

## ➤ java变量分类

### ➤ 按数据类型划分

- 基本数据类型变量

- 引用数据类型变量

### ➤ 按被声明位置划分

- 全局变量：类的内部方法的外部声明的变量

- 局部变量：方法内部或语句块内部声明的变量

# 局部变量的声明和初始化

## ➤ 语法格式

➤ 变量类型 变量名1[=值1][, 变量名2[=值2]...]

## ➤ 声明和初始化例

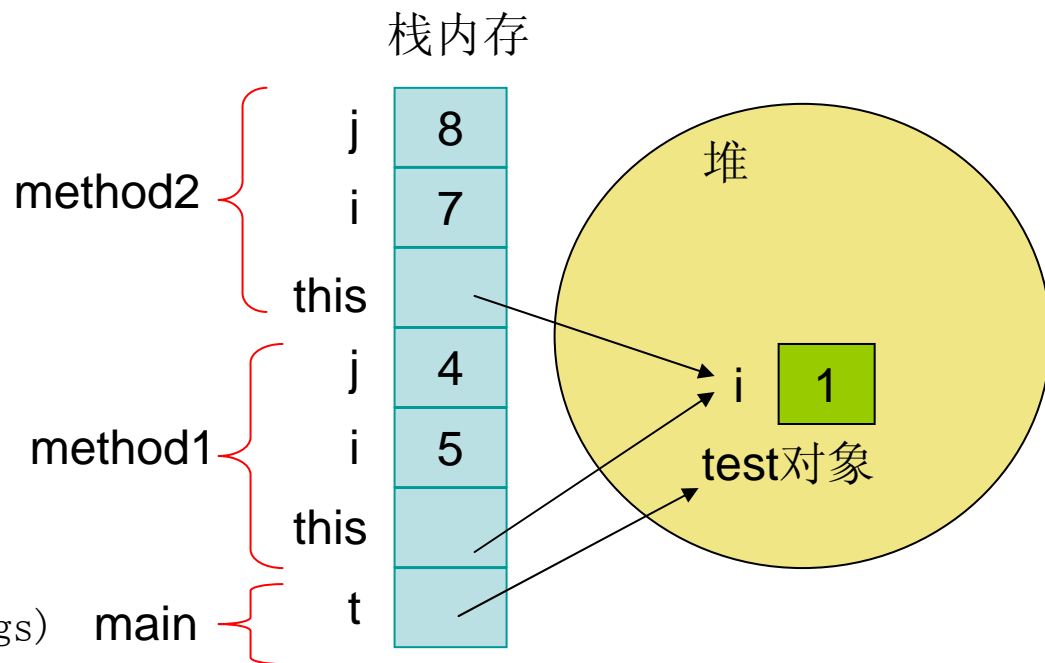
```
public void amethod() {  
    int i;  
    int j = i + 5 ;  
    double d = 3.14;  
    MyDate m;  
    m = new MyDate(22, 7, 1964);  
    System.out.println(m.getYear());  
}
```

# 变量的作用域

- 变量的作用域：
- 局部变量的作用域：在所在的方法或语句块中
  - 在程序调用方法(进入语句块)时，局部变量才被创建并可用，随方法(语句块)的退出，局部变量将被销毁
- 全局变量的作用域：
  - 成员变量依附于对象(局部变量)存在，具有与对象相同的生存期和作用域。

# 变量作用域举例

```
public class Test {  
    private int i=1;  
    public void method1() {  
        int i=4, j=5;  
        this.i= i + j;  
        method2(7);  
    }  
    public void method2(int i) {  
        int j=8;  
        this.i= i + j;  
        System.out.println(this.i);  
    }  
    public static void main(String[] args) {  
        Test t = new Test();  
        t.method1();  
    }  
}
```



# 运算符

算术运算符：+，-，\*，/，%，++，--

关系运算符：>，<，>=，<=，==，!=

布尔逻辑运算符：!，&，|，^，&&，||

位运算符：&，|，^，~，>>，<<，>>>

赋值运算符：= 扩展赋值运算符：+=，-=，\*=，/=

字符串连接运算符：+

# 逻辑运算符

## ➤ 逻辑运算符功能

! --逻辑非      &--逻辑与      |--逻辑或

^ --逻辑异或    &&--短路与      ||--短路或

## ➤ 短路逻辑运算符应用

&& --第一个操作数为假则不判断第二个操作数

|| --第一个操作数为真则不判断第二个操作数

# 逻辑运算符

例如：

val初始值可能为0，1 计算结束val和boolean test的值可能为？

```
boolean test = (val == 0) && (++val == 2);
```

```
boolean test = (val == 0) & (++val == 2);
```

```
boolean test = (val == 0) || (++val == 2);
```

```
boolean test = (val == 0) | (++val == 2);
```



# 赋值运算符

## ➤ 赋值运算符 =

当“=”两侧的数据类型不一致时，可以适用默认类型转换(变量能自动升级到更长的形式, 如int值总可以赋给long变量)或强制类型转换原则进行处理

```
double z=12.414f;
```

```
long l = 100;//默认类型转换
```

```
intsmallval=99L;//非法
```

```
inti = (int)l; //强制类型转换
```

特例:

可以将整型常量直接赋值给byte, short, char等类型变量，而不需要进行强制类型转换，只要不超出其表数范围

```
byte b = 12;//合法
```

```
byte b = 4096;//非法
```

# 不同类型之间的转换

- 自动类型转换
  - ✓ 多种不同类型的数值进行混合运算时，会发生自动类型转换
  - ✓ 转换的原则：向两个运算元中取值范围较大的类型转换
- 强制类型转换
  - ✓ 不进行强制类型转换，就不能将一个值赋给范围较小的类型的变量
  - ✓ 强制转换的语法：（*type*）变量或者（*type*）直接量

# 赋值运算符

- 当把一个引用变量赋给另一个引用变量是，两个变量引用了同一个对象。

举例：

```
class A{}  
class B {  
    public static void main (String [] args) {  
        A a = new A();  
        A c = a;  //c和a引用同一个实例  
    }  
}
```

# 简洁赋值运算符

- $i += 5$  相当于  $i = i + 5$
- $i -= 5$  相当于  $i = i - 5$
- $i *= 5$  相当于  $i = i * 5$
- $i /= 5$  相当于  $i = i / 5$
- $i \% = 5$  相当于  $i = i \% 5$

# 字符串连接运算符

## ➤ 字符串连接运算符：+

➤ "+" 除用于算术加法运算外，还可用于对字符串进行连接操作

```
inti = 300 +5;
```

```
String s = "hello, " + "world!";
```

➤ "+"运算符两侧的操作数中只要有一个是字符串(String)类型，系统会自动将另一个操作数转换为字符串然后再进行连接

```
inti = 300 +5;
```

```
String s = "hello, " + i + "号";
```

```
System.out.println(s); //输出： hello, 305号
```

# 三目运算符

- 简化的逻辑判断语句
- 语法格式： <布尔表达式>? <返回值1>: <返回值2>
- 布尔表达式为**true**返回值1， 为**false**返回值2

# 表达式

- 表达式是符合一定语法规则的运算符和操作数的序列

a

5.0 + a

(a-b)\*c-4

i<30 && i%10 != 0

- 表达式的类型和值

- 对表达式中操作数进行运算得到的结果称为表达式的值

- 表达式的值的数据类型即为表达式的类型

- 表达式的运算顺序

- 首先应按照运算符的优先级从高到低的顺序进行

- 优先级相同的运算符按照事先约定的结合方向进行

# 运算符优先级

结合方向	操作符
R 到 L	++ -- + - ~ ! （数据类型）
L 到 R	* / %
L 到 R	+ -
L 到 R	<< >> >>>
L 到 R	< > <= >= isinstance
L 到 R	== !=
L 到 R	&
L 到 R	^
L 到 R	
L 到 R	&&
L 到 R	
R 到 L	? :
R 到 L	= *= /= %= += -= <<= >>= >>>= &= ^=  =



# 条件分支语句

- 定义：分支语句实现程序流程控制的功能，即根据一定的条件有选择的执行或者跳过一些语句
- Java分支语句分类
  - if-else语句
  - switch语句

# if-else语句语法格式

```
if(boolean类型表达式) {  
    语句或语句块;  
}
```

```
if(boolean类型表达式) {  
    语句或语句块;  
} else if(boolean类型表达式) {  
    语句或语句块;  
} else {  
    语句或语句块;  
}
```

# switch语句语法格式

```
switch(表达式) {  
    case 常量1:  
        语句1;  
        break;  
    case 常量2:  
        语句2;  
        break;  
    .....  
    case 常量N:  
        语句N;  
        break;  
    [  
        default:  
            语句;  
            break;  
    ]  
}
```

# switch语句有关规则

- 表达式expr的返回值必须是下述几种类型之一：int, byte, char, short或枚举常量；
- case子句中的值必须是常量，且所有case子句中的值应是不同的；
- default子句是任选的；
- break语句用来在执行完一个case分支后使程序跳出switch语句块；

# 循环语句

- 循环语句功能
  - 在循环条件满足的情况下，反复执行特定代码
- 循环语句的四个组成部分
  - 初始化部分
  - 循环条件部分
  - 循环体部分
  - 迭代部分
- 循环语句分类
  - for 循环
  - while 循环
  - do/while 循环

# for 循环语句

## ➤ 语法格式

```
for (初始化; 循环条件; 迭代) {  
    循环体  
}
```

## ➤ 例：从1加到100的结果

```
int result = 0;  
for (int i = 1; i <= 100; i++) {  
    result += i;  
}  
System.out.println("result=" + result);
```

# while 循环语句

## ➤ 语法格式

[初始化]

```
while( 循环条件) {
```

```
    循环体
```

```
    [迭代]
```

```
}
```

## ➤ 例：

```
int result = 0, i = 1;
```

```
while (i <= 100) {
```

```
    result += i;
```

```
    i ++;
```

```
}
```

```
System.out.println("result=" + result);
```

# do/while 循环语句

## ➤ 语法格式

[初始化]

do {

    循环体

    [迭代;]

} while( 循环条件);

## ➤ 例:

```
int result = 0, i = 1;
```

```
do{
```

```
    result += i;
```

```
    i ++;
```

```
}while (i <= 100);
```

```
System.out.println("result=" + result);
```



# 遍历数组和集合的循环

➤ 循环的次数等于数组或集合中元素的个数，每次循环都取出一个元素赋给自定义的变量

➤ 语法格式

```
for (元素类型 变量 : 数组或集合对象) {
```

```
.....
```

```
}
```

➤ 例：

```
int[] a = {1, 2, 3, 4};  
for (int item : a) {  
    System.out.println(item);  
}
```

# 特殊流程控制语句

## ➤ break语句

➤ 跳出当前循环或语句块

## ➤ 举例：

.....

```
if (条件为真) {
```

```
    break;
```

```
}
```

.....

# 特殊流程控制语句

➤ continue语句

➤ 跳过本次循环

➤ 举例：

.....

```
if (条件为真) {
```

```
    continue;
```

```
}
```

.....

本章结束

# 第五章 数组

## ➤ 数组概述

- 数组是多个相同类型数据的组合, 实现对这些数据的统一管理
- 数组属于引用类型, 数组型数据是对象, 数组中的每个元素相当于该对象的成员变量
- 数组中的元素可以是任何数据类型, 包括基本类型和引用类型

# 一维数组的声明和创建

## ➤ 声明一维数组

➤ `int[ ] i` 或者 `int i[ ]` 都是合法的，也可以声明引用类型的数组 `Cat[ ] c`，但不能够在声明时指定数组长度 如：

`int[5] i`

## ➤ 创建一维数组

➤ 同对象一样，使用 `new` 关键字，必须在创建时设置数组大小

➤ 例： `i = new int[5]` `c = new Cat[3]`

➤ 数组的长度可以用变量表示

➤ 例： `int length = 5; int i = new int[length];`

# 访问数组元素和长度

- 数组中每个元素都有一个索引（下标），从0开始，可以用 **数组变量名[索引]** 的形式访问数组元素
  - 例： `i[0]`      `c[2]`
- 数组对象的 **length** 属性会返回数组的长度，此属性值只能访问，不可更改
  - 例： `System.out.print(i.length)`

# 数组元素的赋值

- 数组元素在创建数组对象时会被自动赋予默认值
- 可以在创建时直接赋初值
  - 例 `int[] i = {1, 2, 3, 4, 5};`
- 或者用以下方法

```
int[] i = new int[5];
```

```
i[0] = 1;
```

```
i[1] = 2;
```

```
i[2] = 3;
```

```
i[3] = 4;
```

```
i[4] = 5;
```



# 多维数组

## ➤ 二维数组举例

```
int a[][] = { {1, 2}, {3, 4, 0, 9}, {5, 6, 7} } ;
```

## ➤ 分解：

```
int a1[] = {1, 2} ;
```

```
int a2[] = {3, 4, 0, 9} ;
```

```
int a3[] = {5, 6, 7} ;
```

```
int a[][] = {a1, a2, a3} ;
```

## ➤ 结论：Java中多维数组被做为数组的数组处理，并不是必须是一个规则的矩阵形式

# 调整数组大小

➤ 数组的大小在创建后是不可调整的，但是可以通过使用相同的变量引用另外一个全新的数组来改变大小。

➤ 例：

```
int[] i = {1, 2, 3, 4, }; i = new int[5];
```

➤ 注意

➤ 由于使用新的数组，以前数组的元素会全部丢失

# 数组的复制

- 可以使用System对象提供的arraycopy方法直接对数组进行复制

```
int[] a1 = {3, 5, 8, 6, 4}; //数组1
```

```
//创建数组a2元素个数和数组1相同
```

```
int[] a2 = new int[a1.length];
```

```
//将数组a1的所有元素复制到数组a2
```

```
System.arraycopy(a1, 0, a2, 0, a1.length);
```

# 数组实用类：Arrays

➤ 用 `java.util.Arrays` 类可以对数组进行一系列操作

➤ 例

```
int[] a1 = {5, 8, 6, 4, 3};
```

```
//对数组元素进行排序（升序）
```

```
Arrays.sort(a1);
```

```
//返回数组中所有元素，打印[3, 4, 5, 6, 8]
```

```
System.out.println(Arrays.toString(a1));
```

本章结束