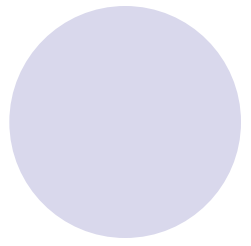
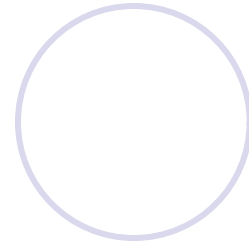
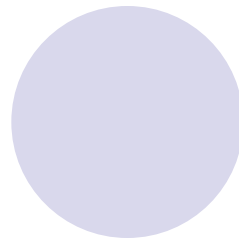
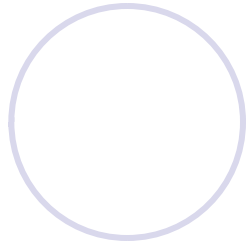
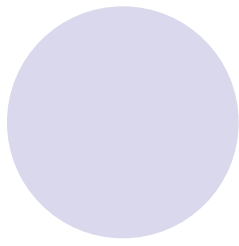


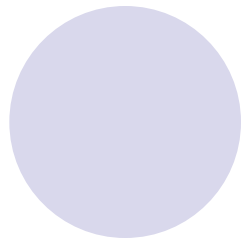
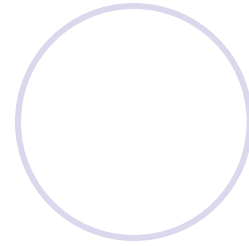
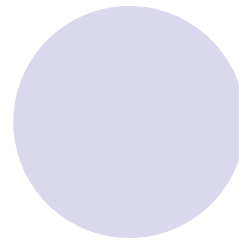
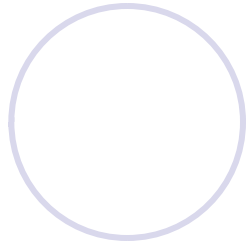
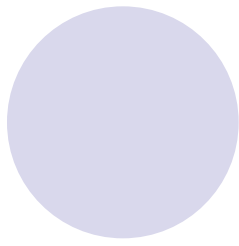
第3章 C#程序设计基础

- 3.1 C#的关键字和标识符
- 1. 关键字
- 关键字是对编译器具有特殊意义的预定义保留标识符。它们不能在程序中用做标识符，除非用一个@前缀。编译器在扫描源程序时，遇到关键字将做出专门的解释。
C#的关键字如表3-1所示。

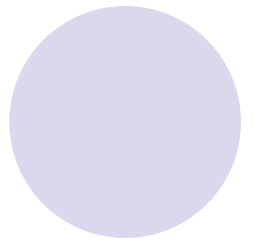
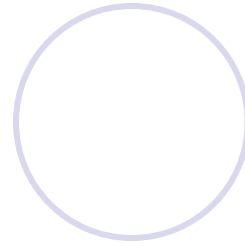
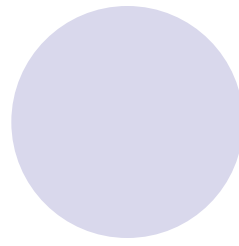
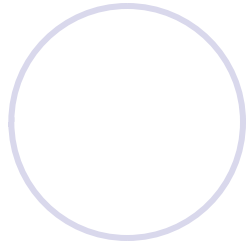
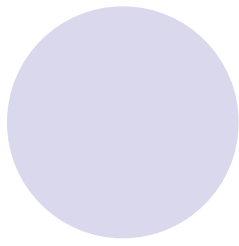
abstract	as	base	bool	break	byte	case
catch	char	checked	class	const	continue	decimal
default	delegate	do	double	else	enum	event
explicit	extern	false	finally	fixed	float	for
foreach	goto	if	implicit	in	int	interface
internal	is	lock	long	namespace	new	null
object	operator	out	override	params	private	protected
public	readonly	ref	return	sbyte	sealed	short
sizeof	stackalloc	static	string	struct	switch	this
throw	true	try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	volatile	void	while



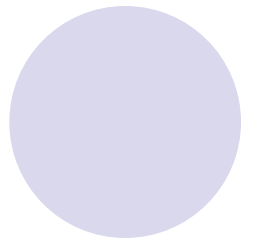
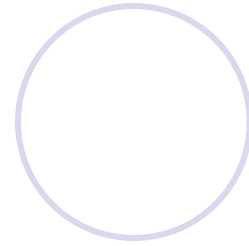
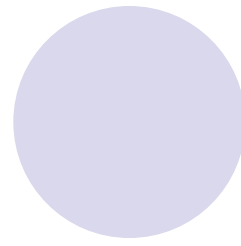
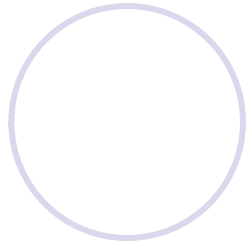
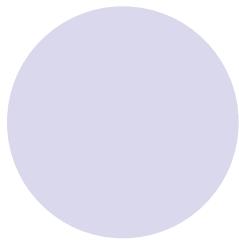
- 下面对部分关键字进行解释：
- **as**：转换操作符，如果转换失败，就返回null。
- **base**：用于访问被派生类或构造中的同名成员隐藏的基类成员。
- **checked**：既是操作符又是语句。确保编译器运行时，检查整数类型操作或转换时出现的溢出。
- **event**：允许一个类或对象提供通知的成员，必须是委托类型。
- **explicit**：定义用户自定义转换操作符的操作符，通常用来将内建类型转换为用户定义类型或反向操作，必须在转换时调用显式转换操作符。



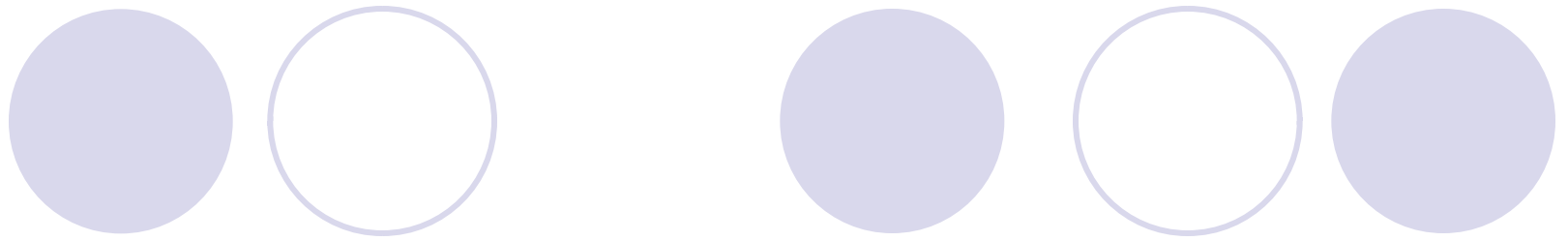
- **extern**: 标识一个将在外部(通常不是C#语言)实现的方法。
- **fixed**: 在一个代码块执行时, 在固定内存位置为变量指派指针。
- **implicit**: 定义用户定义的转换操作符。通常用来将预定义类型转换为用户定义类型或反向操作。隐式转换操作符必须在转换时使用。
- **operator**: 用来声明或重载一个操作符。
- **readonly**: 标识一个变量的值在初始化后不可修改。
- **ref**: 标识一个参数值可能会受影响的参数。
- **sealed**: 防止类型被派生, 防止方法和property被重载。
- **sizeof**: 操作符, 以byte为单位返回一个值类型的长度。



- **stackalloc**: 返回在堆上分配的一个内存块的指针。
- **struct**: 一种值类型，可以声明常量、字段、方法、**property**、索引器、操作符、构造器和内嵌类型。
- **typeof**: 操作符，返回传入参数的类型。
- **unchecked**: 禁止溢出检查。
- **unsafe**: 标注包含指针操作的代码块、方法或类。
- **volatile**: 标识一个可被操作系统、某些硬件设备或并发线程修改的**attribute**。

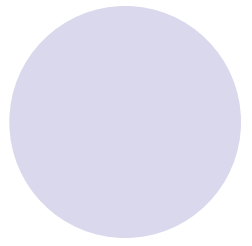
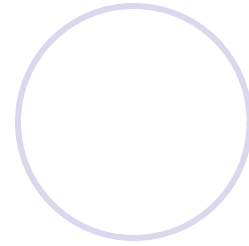
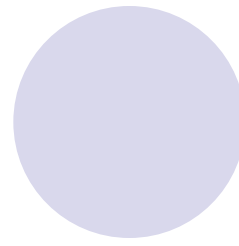
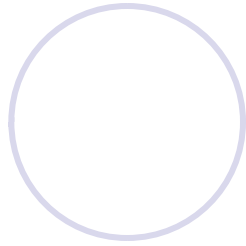
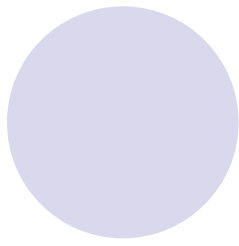


- 2. 标识符
- 标识符在程序设计中的作用是为程序中涉及的对象命名，这些数据对象包括变量、类、对象、方法及文件等。在软件开发中，往往需要对数据对象使用统一的命名规则来约束程序代码的编写，这样可以在软件开发中尽可能减少错误，提高软件开发效率，方便程序员之间的交流和软件系统的维护。
- 在**C#**程序中，标识符规则完全符合**Unicode**标准推荐的规则，另外，标识符的命名必须遵守下列规则：

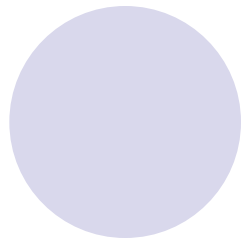
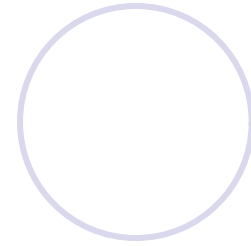
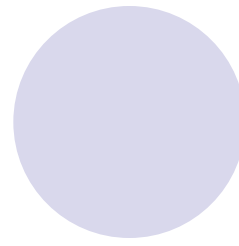
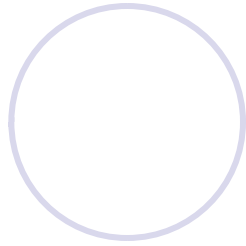
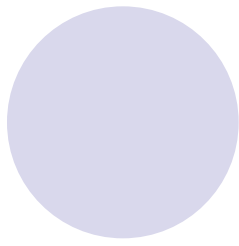


- (1) 第一个字符必须是英文字母或下画线(实际上也可以是汉字、希腊字母、俄文字母等其他Unicode字符, 但不推荐, 通常不这样使用)。
- (2) 从第二个字符开始, 可以使用英文字母、数字或下画线, 但不能包含空格、标点符号、运算符等字符。
- (3) 不能与关键字重名, 但允许以“@”字符作为前缀, 以使关键字能够用做标识符。
- (4) 长度不能超过**255**个字符。

- 在实际应用中，为了改善程序的可读性，标识符最好使用便于记忆的英文单词。如**FirstName**表示名，**LastName**表示姓。
- 目前，软件开发中使用较多的标识符命名样式主要有以下几种：
- (1) 骆驼式(**Camel**)命名法
- 定义：除第一个单词的首字母小写外，其他单词的首字母都大写，如**userFirstName**。
- (2) 匈牙利命名法
- 定义：标识符的名字以一个小写字母开头作为前缀，标识出变量的作用域、类型等，前缀之后是首字母大写的单词或多个单词组合。
- 前缀符号多个同时使用，顺序是先**m_**(成员变量)，再指针，再简单数据类型，再其他。如**m_lpszStr**，表示指向一个以0字符结尾的字符串的长指针成员变量。



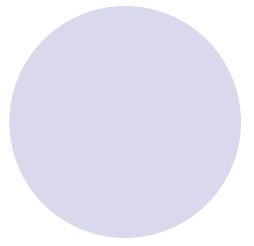
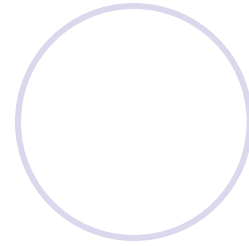
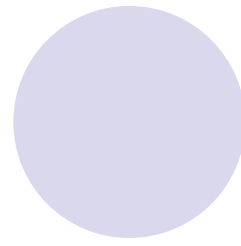
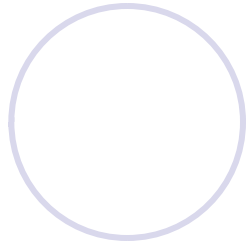
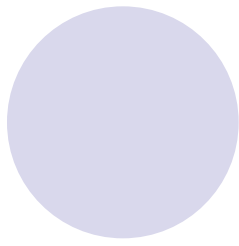
- (3) 帕斯卡(Pascal)命名法
- 定义：首字母大写的一个或多个单词组合，如 **UserNameList**。
- (4) 符号连接命名法
- 定义：最具代表性的两种符号连接命名法是下画线(_)命名法和连字符(-)命名法，分别使用下画线(_)和连字符(-)来分隔单词，如**left_side**或**left-side**。
- (5) 语义化命名法
- 定义：将语义部分包含在命名中，类似于匈牙利命名法。



- 下面的标识符可以作为合法的标识符使用：
- **B Y2 Ppi first FirstName @long _kk**
- 下面的标识符不是合法的标识符：
- **2Y a + b m, n int (a)**

3.2 C#的数据类型

- 为便于表示不同类型的信息，并尽可能避免存储空间的浪费，提高程序运行效率，**C#**提供了多种数据类型。这些数据类型分为两大类：值类型和引用类型。
- 值类型包括简单类型、结构类型、枚举类型。
- 引用类型主要包括类(class)、数组(array)、接口(interface)、委托(delegate)。



- 3.2.1 值类型

- 所谓值类型就是一个包含实际数据的量。即当定义一个值类型的变量时，**C#**会根据它所声明的类型，以堆栈方式分配一块大小相适应的存储区域给这个变量，随后对这个变量的读或写操作就直接在这块内存区域进行。**C#**中的值类型包括：简单类型、枚举类型和结构类型。

- 图3.1 值类型操作示意图
- 例如:
- `int a=20;`//分配一个32位内存区域给变量a，并将20放入该内存区域
- `a=a+5;` //从变量a中取出值，加上5，再将计算结果赋给a
- 图3.1给出了该值类型的操作示意。

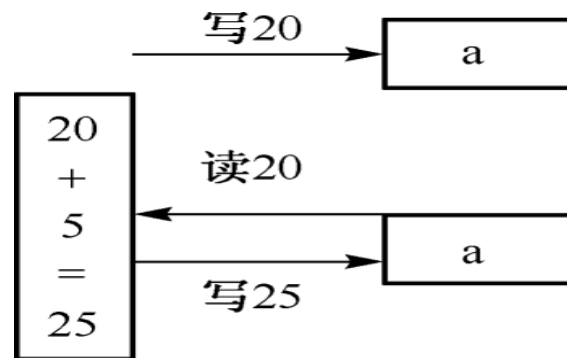
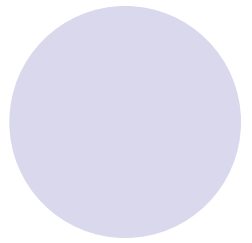
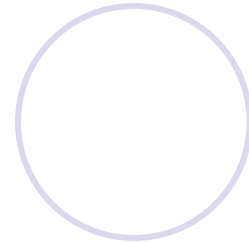
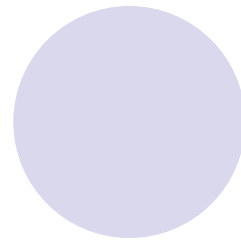
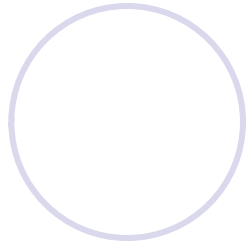
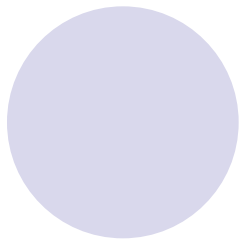


图3.1 值类型操作示意图

1. 简单类型


- 简单类型是系统预置的，可分为整数类型、布尔类型、字符类型、实数类型。简单类型一共有13个数值类型，如表3-2所示。

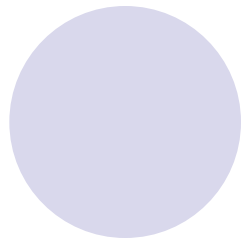
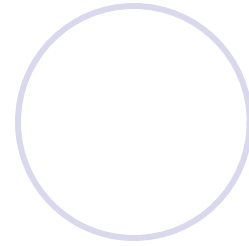
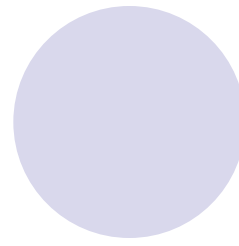
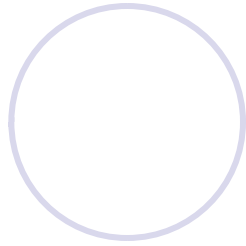
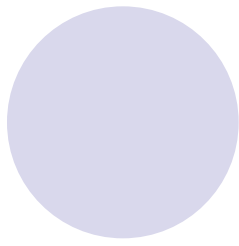
C#关键字	.NET CTS类型名	描 述	范围和精度
bool	System.Boolean	逻辑值(真或假)	true、false
sbyte	System.SByte	8位有符号整数类型	-128~127
byte	System.Byte	8位无符号整数类型	0~255
short	System.Int16	16位有符号整数类型	-32768~32767
ushort	System.UInt16	16位无符号整数类型	0~65535
int	System.Int32	32位有符号整数类型	-2147483648~2147483647
uint	System.UInt32	32位无符号整数类型	0~4294967295
long	System.Int64	64位有符号整数类型	-9223372036854775808~ 9223372036854775807
ulong	System.UInt64	64位无符号整数类型	0~18446744073709551615
char	System.Char	16位字符类型	所有的Unicode编码字符
float	System.Single	32位单精度浮点类型	$\pm 1.5 \times 10^{-45} \sim \pm 3.4 \times 10^{38}$
double	System.Double	64位双精度浮点类型	$\pm 5.0 \times 10^{-324} \sim \pm 3.4 \times 10^{308}$
decimal	System.Decimal	128位高精度十进制数 类型	$\pm 1.0 \times 10^{-28} \sim \pm 7.9 \times 10^{28}$



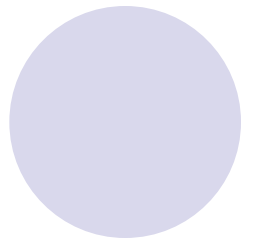
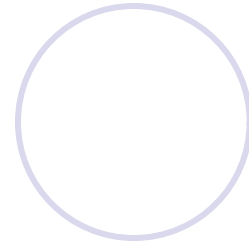
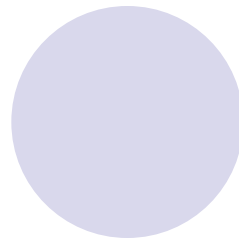
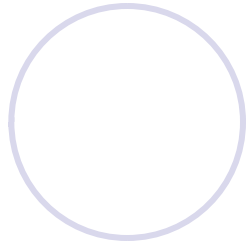
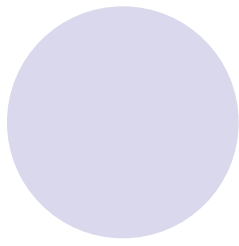
- 2. 枚举类型

- 枚举类型由一组命名常数组成，这组命名常数称为枚举列表；每个枚举类型都具有一个底层基本类型。底层基本类型必须是内置的有符号或无符号整数类型(如int16、int32、int64)，除了char之外的所有整数类型都可以作为枚举类型的基本类型，枚举元素的默认类型为int，在默认情况下，第一个元素的值为0，后继的元素依次递增1。例如，在下面的枚举类型color1中，red=0、green=1。枚举类型使用enum关键字声明，该类型的声明形式如下：

- 
- `enum 枚举类型名 [: 基本数据类型] { 枚举常量列表 }`
 - `enum color1 { red, green, blue } //`
`red=0, green=1, blue=2`
 - `enum color2 : byte { red, green=2, blue } //`
`red=0, green=2, blue=3`
 - 在C#语言中，枚举不能作为一个整体被引用，只能使用“枚举类型名.枚举成员名”的方式访问枚举中的个别成员。枚举成员本质上是一个常量，因而不允许向其赋值，只能被读取，而且应该通过强制类型转换才能将其转换为基本类型的数据。声明一个名称为rd的、类型为color1的变量：
 - `color1 rd=color1.red;`
 - `int a=(int)rd;`



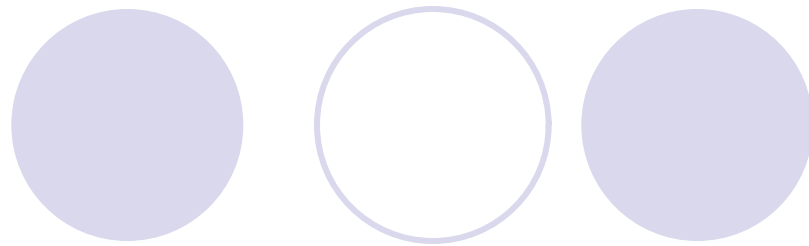
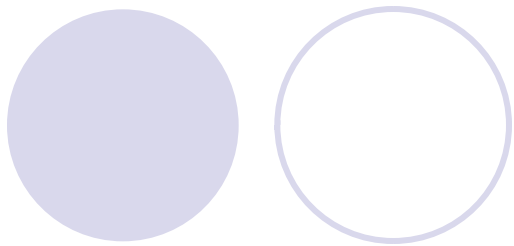
- 3. 结构类型
- 把一系列相关的变量组织成一个单一实体的类型称为结构类型，每个变量称为结构的成员。
- 【例3-1】 将一个点的x坐标和 y坐标组成一个点的结构类型。结构是使用struct关键字定义的，定义一个点结构类型point的代码如下：
- struct point //声明结构
- {
- public int x,y; //结构成员的声明
- public point(int p1,int p2) //定义构造函数
- {
- x=p1;
- y=p2;
- }
- }



- 创建结构实例的方法：
- `point a`
- `a.x=3;` //引用结构的x成员
- `a.y=4;`
- 结构和枚举一样，不能作为一个整体被引用，程序中只能以“结构名.结构成员名”的形式访问结构中的成员，并对其进行读/写操作。

3.2.2 引用类型

- 引用类型包括类(class)、接口(interface)、数组、委托(delegate)、object和string。其中object和string是两个比较特殊的类型。object是C#中所有类型(包括所有的值类型和引用类型)根类。string类型是一个从object类直接继承的密封类型(不能再被继承)，其实例表示Unicode字符串。
- 引用类型的变量不存储它们所代表的实际数据，而是存储实际数据的引用。引用类型分两步创建：首先在堆栈上(栈内存)创建一个引用变量，然后在堆上创建对象本身，再把这个堆内存的句柄(也是内存的首地址)赋给引用变量。例如：



- `string s1, s2;`
- `s1="zhaomin"; s2 = s1;`
- 图3.2 引用类型赋值示意图
- 其中，`s1`、`s2`是指向字符串的引用变量，`s1`的值是字符串“`zhaomin`”存放在堆内存的地址，这就是对字符串的引用。两个引用型变量之间的赋值，使得`s2`、`s1`都是对“`zhaomin`”的引用，如图3.2所示。

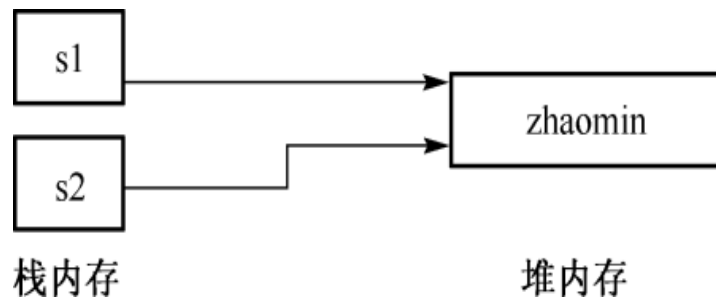
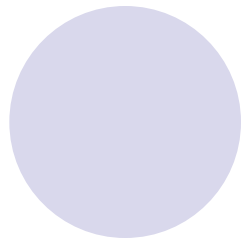
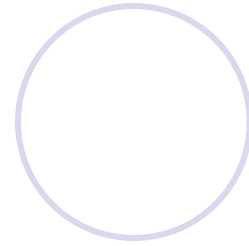
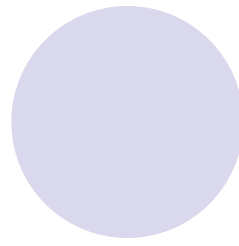
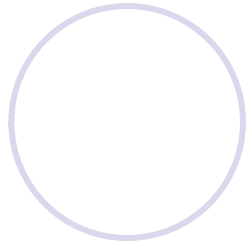
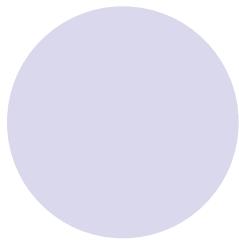


图3.2 引用类型赋值示意图



- 引用类型(**reference type**)的变量在堆栈中存储对数据(对象)的引用(地址), 数据(对象)存储在托管运行环境管理的堆(**Heap**)中。
- 堆与引用类型: 两个变量可能引用同一个对象, 因此对一个变量的操作可能影响另一个变量所引用的对象。
- **【例3-2】** 变量**val1**、**val2**为值类型, **ref1**、**ref2**为引用类型, 比较其运行结果。其内存分配示意图如图3.3所示。

- using System;
- using System.Collections.Generic;
- using System.Text;
- namespace ConsoleApplication1
- {
- class pppp
- { public int Value = 0; }
- class TestType
- {
- static void Main(string[] args)
- {
- int val1 = 0; int val2 = val1; val2 = 123;
- pppp ref1 = new pppp ();
- pppp ref2 = ref1; ref2.Value = 123;
- Console.WriteLine("Values:{0},{1}",val1,val2);
- Console.WriteLine("refs:{0},{1}",ref1.Value,ref2.Value);
- Console.ReadLine();
- }
- }
- }

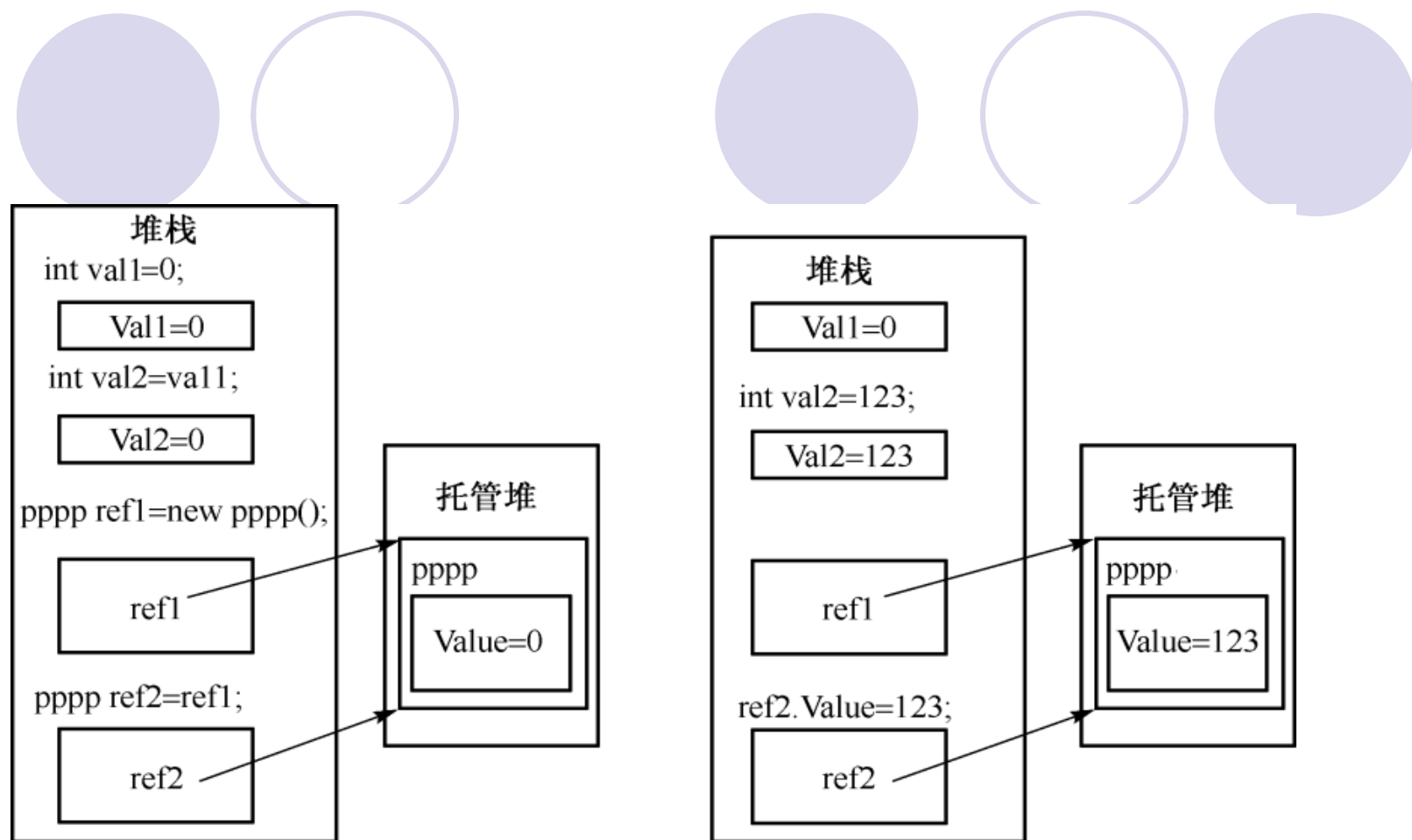
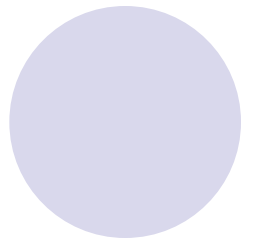
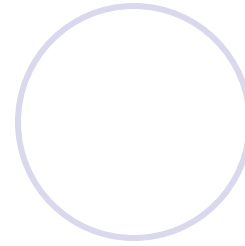
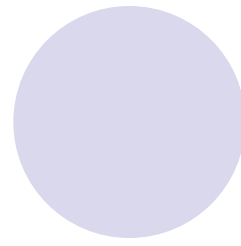
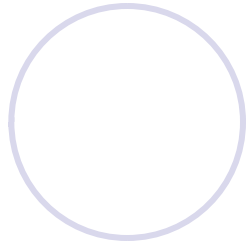
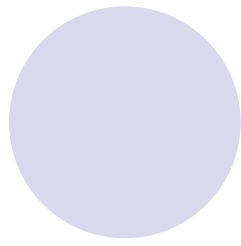


图3.3 内存分配示意图

- 
- **val1、val2**的数据类型是值类型，存放在堆栈中。而**ref1、ref2**是属于**pppp**类的对象，属于引用类型，在堆栈中给**ref1、ref2**分配了内存，但在该内存中存放的是地址。**new pppp();**语句完成的功能是在托管堆里分配一块内存地址，**pppp ref1 = new pppp();**语句是把刚刚在托管堆里分配的内存地址的首地址存到在堆栈中给**ref1**分配的地址的内存中。**pppp ref2 = ref1;**语句表示引用类型的对象**ref2**和**ref1**共同指向了托管堆里的同一块内存的首地址。**ref2.Value = 123;**语句表示**ref2**所指向的托管堆中的内存所存储的**Value**成员的值发生了变化，由于**ref1** 和**ref2**指向托管堆中的同一块地址，因此**ref1.Value**的值也发生了变化。

3.2.3 值类型与引用类型关系

- **C#**的类型系统是统一的，因此任何类型的值都可以作为对象进行处理。**C#**中的所有类型都直接或间接地继承自**object**类型，并且**object**类是所有类型的最终基类。通过将引用类型的值视为**object**类型，可以很方便地将其作为对象进行处理。
- 装箱可以把值类型与引用类型的值赋给**object**类型变量，**C#**用“装箱”和“拆箱”来实现值类型与引用类型之间的转换。



- 装箱就是将值类型包装成引用类型的处理过程。当一个值类型被要求转换成一个 **object** 对象时，装箱操作自动进行，首先创建一个对象实例，然后把值类型的值复制到这个对象实例，最后由 **object** 对象引用这个对象实例。装箱后的 **object** 对象中的数据位于托管堆中。

- 【例3-3】 装箱操作实例。装箱机制如图3.4所示。

- using System;
- class test
- {
- public static void Main ()
- { int x = 121;
- object obj1=x; //装箱操作
- x = x+10; //改变x的值， obj1的值并不会随之改变
- Console.WriteLine("x={0},obj1={1}",x,obj1);//x=
- 131,obj1=121
- }
- }

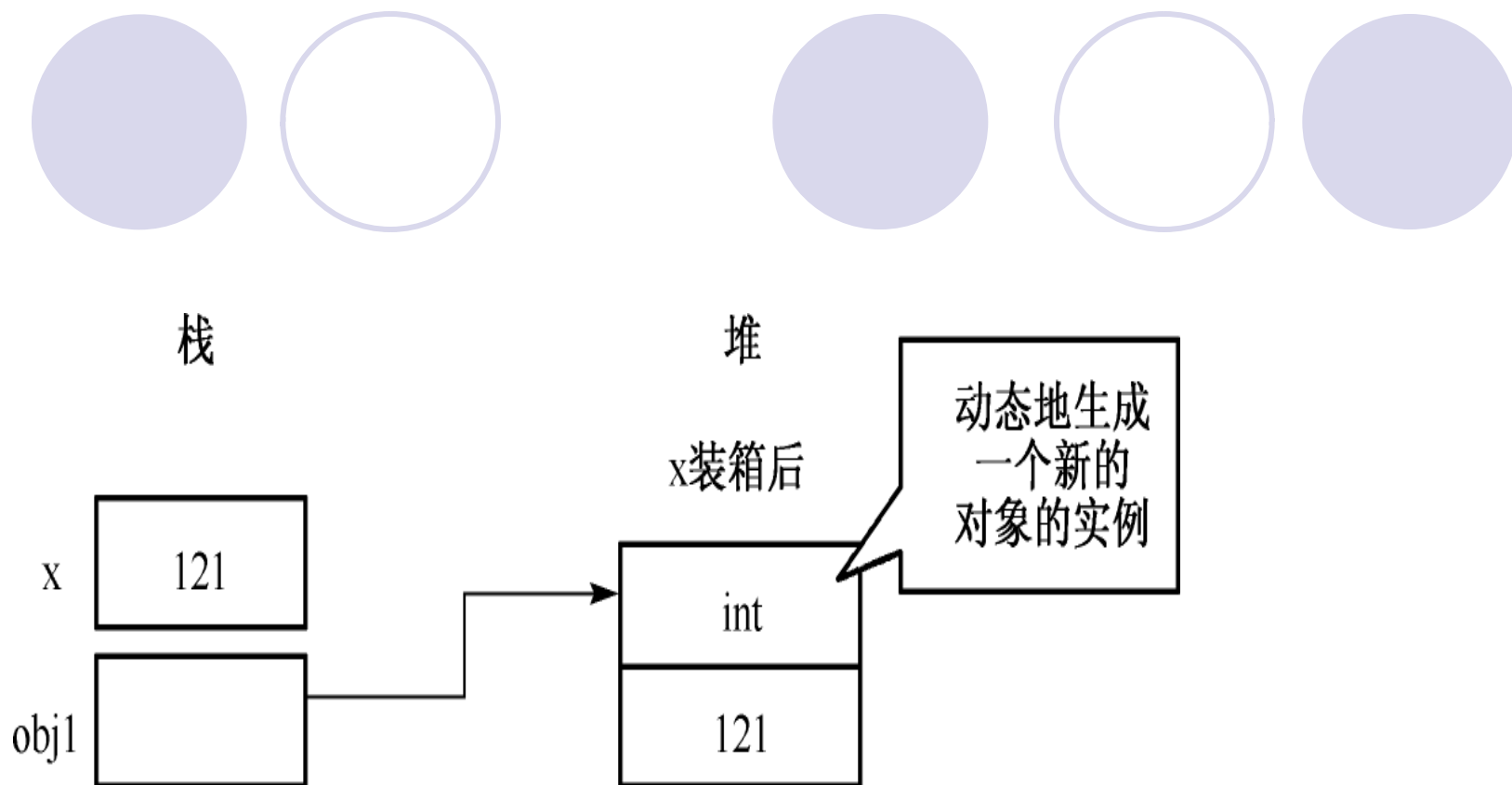
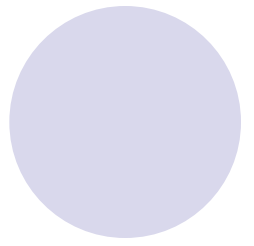
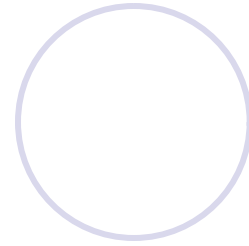
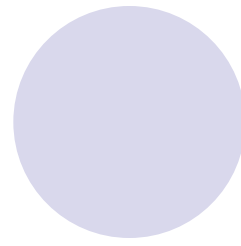
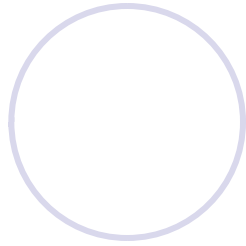
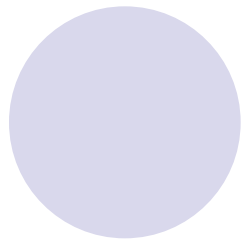


图3.4 装箱机制

- 拆箱操作与装箱相反，是将一个object类型转换成值类型。首先检查由object引用的对象的实例值类型的包装值，然后把实例中的值复制到值类型的变量中。
- 【例3-4】 装箱与拆箱实例。
- using System;
- class test
- {
- public static void Main ()
- { int x = 121, y;
- object obj1=x; // 装箱操作
- x = x+10; // 改变x的值，此时obj1的值并不会随之改变
- y = (int) obj1; // 拆箱操作，必须进行强制类型转换
- Console.WriteLine (" x= {0},obj1={1}" , x,obj1);
- // x=131 ,obj1=121
- }
- }



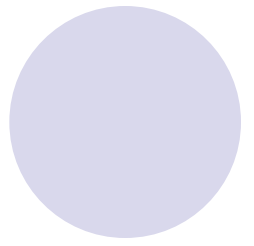
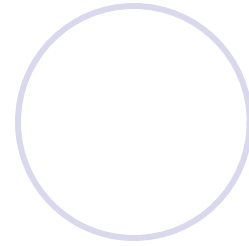
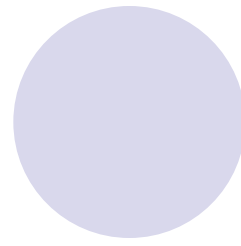
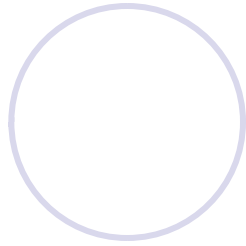
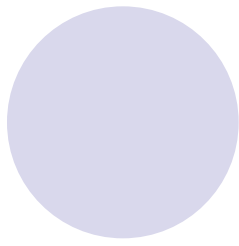
- 注意： 当一个装箱操作把值类型转换为一个引用类型时，不需要显式地强制类型转换；而拆箱操作把引用类型转换到值类型时，由于它可以强制转换到任何可以相容的值类型，所以必须显式地强制类型转换。

3.3 C#中的变量和常量

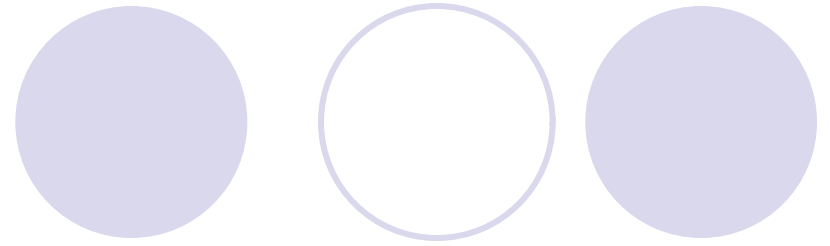
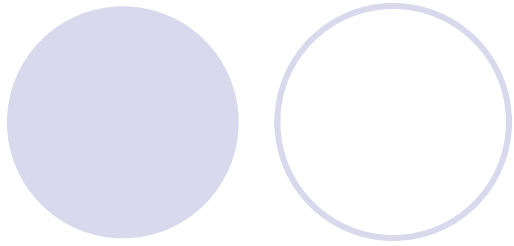
● 3.3.1 变量

- 程序要对数据进行读/写运算等操作，当需要保存特定的值或计算结果时就需要用到变量，在变量中可以存储各种类型的信息。
- 在计算机中变量代表存储地址，变量类型决定了存储在变量中的数值的类型。**C#**是一种安全类型语言，变量必须具有适当的数据类型。
- **C#**中的变量必须先声明后使用。声明变量包括变量的名称、数据类型，以及必要时指定变量的初始值。标识符是变量名的符号。

- 1. 变量声明格式
- 类型 变量名;
- 或
- 类型 变量名1[=初值] [, ..., 变量名n[=初值];
- 2. 变量的分类
- 在C#语言中，变量分为7类：静态变量、非静态变量(或实例变量)、数组元素、值参数、引用参数、输出参数和局部变量。
- (1) 静态变量
- C#静态变量使用**static**修饰符进行声明，一旦静态变量所属的类被装载，直到包含该类的程序运行结束时，它将一直存在。静态变量的初始值就是该变量类型的默认值，为便于定义赋值检查，静态变量最好在定义时赋值，如**static int x=10**。调用时无须实例化就可以直接调用(通过类进行访问)。



- (2) 非静态变量或实例变量
- 不带有 **static** 修饰符声明的变量称做非静态变量(或实例变量)。在类被实例化时创建, 通过对象进行访问, 同一个类的不同实例的同一非静态变量可以是不同的值。
- (3) 数组元素
- 数组元素也是变量的一种, 该变量随数组实例的存在而存在。每个数组元素的初始值是该数组元素类型的默认值。



- (4) 局部变量
- 局部变量是指在一个独立的程序块(在“{”和“}”之间的代码称为程序块)、一个**for**语句、**switch**语句或**using**语句中声明的变量，它只在该范围中有效。当程序运行到这一范围时该变量即开始生效，程序离开这一范围时变量就失效了。与其他变量类型不同的是，局部变量不会自动被初始化，所以也就没有默认值。
- (5) 值参数、引用参数、输出参数在后续章节介绍

- 【例3-5】 在下面的代码中，指出所有变量属于什么类型的变量。

- ```
class bianliangtype
{
 public static int x;
 int y;
 void ED(int[] v, int a, ref int b, out int c)
 {
 int i = 1;
 c = a + b;
 }
}
```

- 其中，**x**是静态变量，**y**是实例变量，**v[0]**是数组元素，**a**是值参数，**b**是引用参数，**c**是输出参数，**i**是局部变量

## 3.3.2 常量

- 常量，就是在程序运行期间其值不会改变的量。使用常量可以提供程序代码的可读性，并使代码更易于维护。常量是有意义的名称，用于代替在应用程序的整个执行过程都保持不变的数字或字符串。
- 要定义一个常量，其一般格式为：
- **[属性][修饰符] const 数据类型 常量名=表达式;**
- 属性、修饰符、数据类型和常量名的要求与定义变量名的规则相同。但为使常量易于识别，常常将常量名大写，程序中也是通过常量名来访问该常量的。表达式的计算结果是所定义常量的值。例如 **const PI=3.14159**。在程序中，常量只能被赋予初始值，在程序运行的过程中不允许再改变。定义常量时，其表达式中的运算对象只能是常量和常数，不能存在变量。

- 整型常量：即整常数。**C#**整常数可用以下两种形式表示：十进制整数；十六进制整数。
- 实型常量：实数在**C#**语言中又称浮点数。实数有两种表示形式：十进制数形式；指数形式。
- 字符型常量：字符数据类型**char**用来处理Unicode字符。Unicode字符是16位字符，用于表示世界上多数已知的书面语言。**char**变量以无符号16位数字的形式存储，取值范围为0~65535。每个数字代表一个Unicode字符。
- **C#**字符常量是用单引号括起来的一个字符。如：**'a'、'x'、'D'、'Q'、'\$'、'3'**都是字符常量。
- **C#**还允许用一种特殊形式的字符常量，就是一个以**"\"**开头的字符序列，称之为转义字符。例如，**'\n'**表示一个换行符，代表“换行”。常用转义符如表3-3所示

表3-3 常用转义符

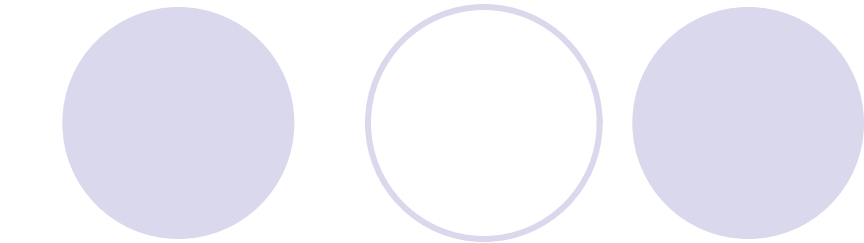
| 转义序列 | 字符名称 | Unicode编码 |
|------|------|-----------|
| \'   | 单引号  | 0x0027    |
| \"   | 双引号  | 0x0022    |
| \\   | 反斜杠  | 0x005C    |
| \0   | 空    | 0x0000    |
| \a   | 警报   | 0x0007    |
| \b   | 退格符  | 0x0008    |
| \f   | 换页符  | 0x000C    |
| \n   | 换行符  | 0x000A    |
| \r   | 回车   | 0x000D    |
| \t   | 水平制表 | 0x0009    |
| \v   | 垂直制表 | 0x000B    |

## 3.4 运算符和表达式

- 3.4.1 运算符
- 按运算功能来分，基本的运算符可以分为：算术运算符、关系运算符、逻辑运算符、位运算符、赋值运算符、条件运算符、其他(分量运算符“.”、下标运算符“[ ]”等)。
- 1. 算术运算符
- 算术运算符(见表3-4)作用的操作数类型可以是整型也可以是浮点型

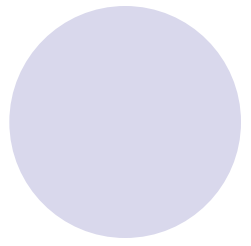
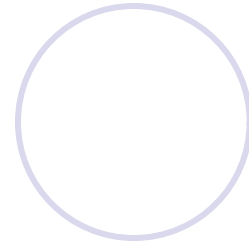
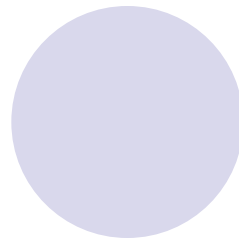
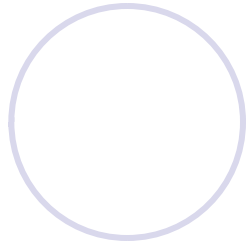
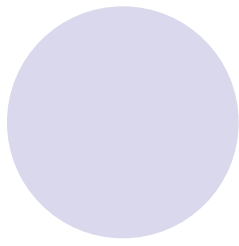


表3-4 C#算术运算符



| 运 算 符 | 运 算 | 表达式示例           | 优 先 级 |
|-------|-----|-----------------|-------|
| +     | 加法  | $x + y$         | 4     |
| -     | 减法  | $x - y$         | 4     |
| %     | 求模  | $x \% y$        | 3     |
| *     | 乘法  | $x * y$         | 3     |
| /     | 除法  | $x / y$         | 3     |
| +     | 一元加 | $+x$            | 2     |
| -     | 一元减 | $-x$            | 2     |
| ++    | 自增  | $x++ \quad ++x$ | 1     |
| --    | 自减  | $x-- \quad --x$ | 1     |

- “+、-、\*、/”运算与一般代数意义及其他语言相同，但需要注意的是：当“/”作用到的两个操作数都是整型数据类型时，其计算结果也是整型。
- (1) 加法运算符的多重作用
- 在C#语言中，根据两个操作数的类型特点，加法运算符具有多重作用。
  - 两个操作数均为数字，相加的结果为两个操作数之和；
  - 两个操作数均为字符串，把两个字符串连接在一起；
  - 两个操作数分别为数字和字符串，则先把数字转换成字符串，然后连接在一起；
  - 两个操作数分别为数字和字符，则先把字符转换成Unicode代码值，然后求和。如 $11+'a'$ ，结果等于108，'a'的ASCII码值是97， $11+97=108$
- 算术运算符的优先级如表3-4所示

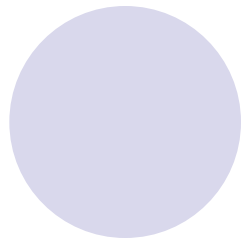
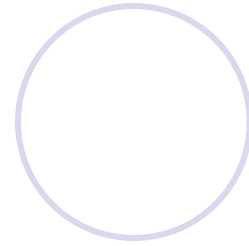
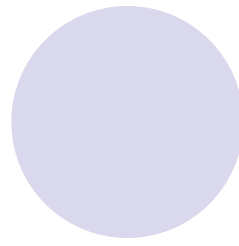
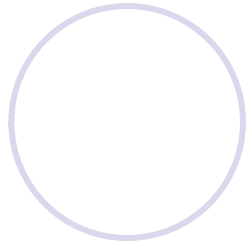
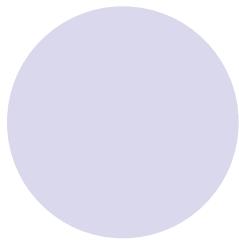


- 自增和自减运算都有一个操作数，有如下几种形式：
- **x++**：操作数**x**的值先被引用，然后加**1**。
- **++x**：操作数**x**的值先加**1**，然后被引用。
- **x--**：操作数**x**的值先被引用，然后减**1**。
- **--x**：操作数**x**的值先减**1**，然后被引用。
- **b=100;**
- **a=++b;** //相当于**b=b+1;a=b;** 结果 **a=101, b=101**
- **b=100;**
- **a=b++;** //相当于**a=b; b=b+1;** 结果 **a=100, b=101**

- (2) 求模运算
- 求模运算本质上也是一种除法运算，只不过它舍弃商而把小于除数的未除尽部分(即余数)作为运算结果，又称取余运算。求模运算结果的符号取决于左操作数的符号。即左操作数为正数，求模结果为正；左操作数为负数，求模结果为负。
- $3 \% 7$       //结果等于3
- $-7 \% -3$       //结果等于 -1
- 2. 关系运算符和类型测试运算符
- 关系运算符和类型测试运算符是二元运算符。关系运算符用于将两个操作数的大小进行比较。若关系成立，则比较结果为**True**，否则为**False**。关系运算符的操作数可以是数值型、字符型和枚举型。表3-5列出了C#中的关系运算符和类型测试运算符。

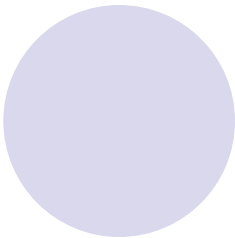
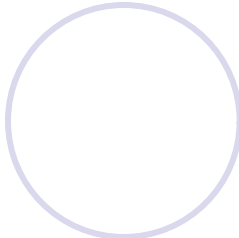
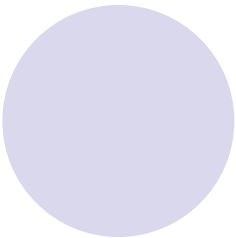
# 表3-5 关系运算符和类型测试运算符

| 关系运算符               | 类型测试关系                     | 表达式示例                    |
|---------------------|----------------------------|--------------------------|
| <code>==</code>     | 相等                         | <code>x == y</code>      |
| <code>!=</code>     | 不相等                        | <code>x != y</code>      |
| <code>&lt;</code>   | 小于                         | <code>x &lt; y</code>    |
| <code>&gt;</code>   | 大于                         | <code>x &gt; y</code>    |
| <code>&lt;=</code>  | 小于或等于                      | <code>x &lt;= y</code>   |
| <code>&gt;=</code>  | 大于或等于                      | <code>x &gt;= y</code>   |
| <code>x is T</code> | 数据x是否属于类型T                 | <code>x is int</code>    |
| <code>x as T</code> | 返回转换类型T的x，如果转换不能进行，则返回null | <code>X as object</code> |

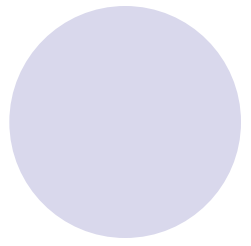
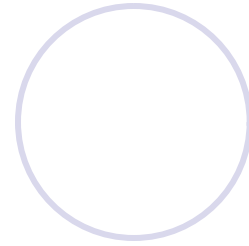
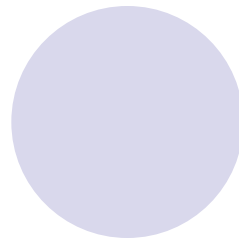
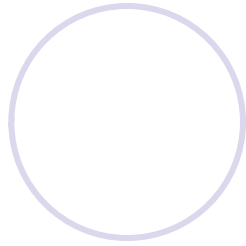
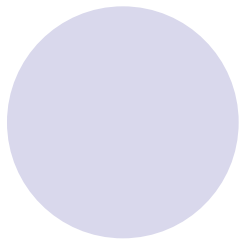


- 注意：关系运算符的优先级相同。
- 对于两个预定义的数值类型，关系运算符按照操作数的数值大小进行比较。
- 对于string类型，关系运算符比较字符串的值，即按字符的ASCII码值从左到右一一比较：首先比较两个字符串的第一个字符，其ASCII码值大的字符串大；若第一个字符相等，则继续比较第二个字符，以此类推，直至出现不同的字符为止。
- 对于string以外的引用类型，如果两个操作数引用同一个对象，则“==”返回True；如果两个操作数引用不同的对象，则“!=”返回True。
- System.Int32和int是相同的数据类型。
- 3. 逻辑运算符

逻辑运算符如表3-6所示。  
表3-6 逻辑运算符



| 逻辑运算符 | 运 算  | 优 先 级 |
|-------|------|-------|
| !     | 逻辑非  | 1     |
| &     | 逻辑与  | 2     |
| ^     | 逻辑异或 | 3     |
|       | 逻辑或  | 4     |
| &&    | 短路与  | 5     |
|       | 短路或  | 6     |



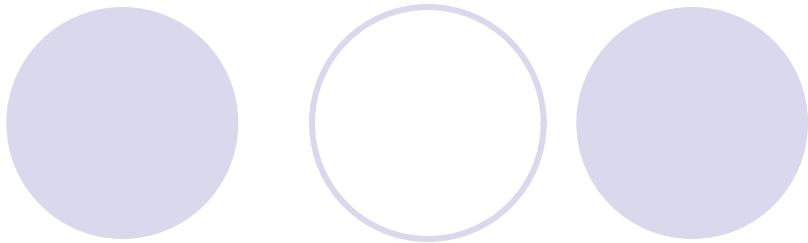
- **!(非):** 唯一的单目逻辑运算符。结果是操作数原有逻辑值的反值。
- **&(与):** 只有左、右操作数的值都为**True**时，结果为**True**，否则为**False**。
- **|(或):** 左、右操作数只要有一个为**True**，结果即为**True**。仅当左、右操作数的值均为**False**时，结果才为**False**。
- **^(异或):** 当左、右操作数的值相同时，结果为0，否则为1。
- **&&和||:** 运算符“&&”和“||”的操作结果与“&”和“|”一样，但它们的短路特征，使代码的效率更高。所谓短路就是在逻辑运算的过程中，如果计算第一个操作数时就能得知运算结果，就不再计算第二个操作数。



#### 4. 位运算符

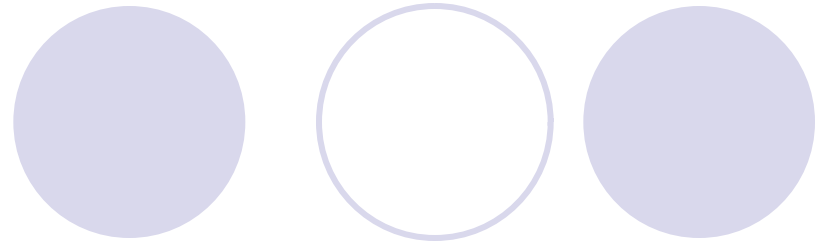
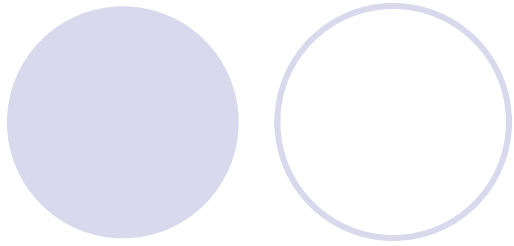
位运算符如表3-7所示。

表3-7 位运算符



| 位运算符 | 运 算      | 表达式示例         | 结 果          | 优 先 级 |
|------|----------|---------------|--------------|-------|
| ~    | 按位取反     | ~0xf8         | 0xffffffff07 | 1     |
| <<   | 左移       | 0x2<<1        | 0x04         | 2     |
| >>   | 右移       | 0xffffffff>>1 | 0x7fffffff   | 2     |
| &    | 按位逻辑“与”  | 0xf7 & 0x3f   | 0x37         | 3     |
| ^    | 按位逻辑“异或” | 0xf8^0x5f     | 0xa7         | 4     |
|      | 按位逻辑“或”  | 0xf8 0x5f     | 0xff         | 5     |

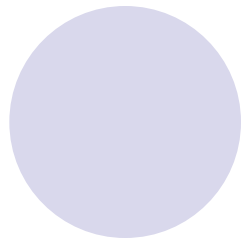
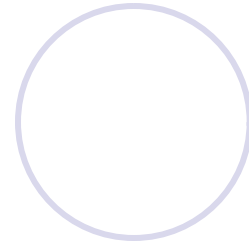
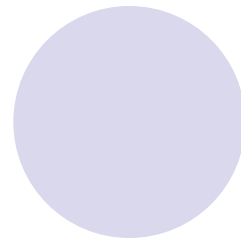
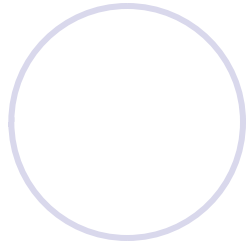
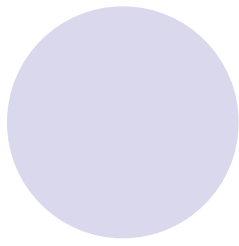
- $\sim$ (按位取反): 将二进制数的各位取原有值的反值。即原来为0, 取反为1; 原来为1, 取反为0。
- $\&$ (对应位“与”): 只有左、右操作数对应位的值都为1时, 结果为1, 否则为0。
- $|$ (对应位“或”): 左、右操作数只要对应位有一个为1, 结果即为1。仅当左、右操作数的值均为0时, 结果才为0。
- $\wedge$ (对应位“异或”): 当左、右操作数对应位的值相同(即都为1或0)时, 结果为0, 否则为1。
- $\ll$ (左移): 将二进制操作数的各位向左移若干位, 相当于逐次乘2的操作。
- $\gg$ (右移): 将二进制操作数的各位向右移若干位, 相当于逐次除2的操作。



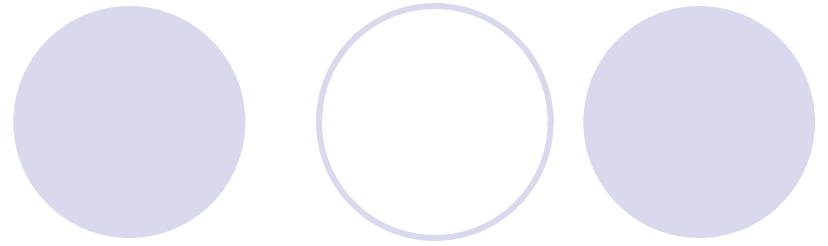
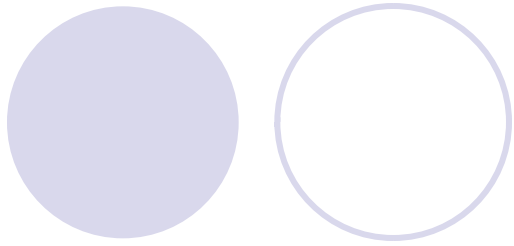
- 5. 赋值运算符
- 简单赋值语句形式如下：
- 变量名 = 表达式;
- 如 `int a;`
- `a=10;`
- 复合赋值语句如表3-8所示

表3-8 复合赋值语句

| 赋值运算符 | 运 算  | 示 例      | 等效于                   |
|-------|------|----------|-----------------------|
| +=    | 加法赋值 | $x+=y$   | $x = x + y$           |
| -=    | 减法赋值 | $x-=y$   | $x = x - y$           |
| *=    | 乘法赋值 | $x*=y+3$ | $x = x * (y+3)$       |
| /=    | 除法赋值 | $x/=y$   | $x = x / y$           |
| %=    | 取模赋值 | $x\%=y$  | $x = x \% y$          |
| <<=   | 左移赋值 | $x<<=y$  | $x = x << y$ , 即x左移y位 |
| >>=   | 右移赋值 | $x>>=y$  | $x = x >> y$          |
| &=    | 与赋值  | $x\&=y$  | $x = x \& y$          |
| =     | 或赋值  | $x =y$   | $x = x   y$           |
| ^=    | 异或赋值 | $x^=y$   | $x = x ^ y$           |



- 6. 条件运算符
- 条件运算符“?:”是C#中唯一一个三元运算符，其形式为：
- 逻辑表达式? 表达式1: 表达式2
- 使用条件运算符，其中逻辑表达式的运算结果必须是一个bool类型值，表达式1和表达式2可以是任意数据类型，但它们返回的数据类型必须一致。
- 条件运算符的运算过程是：首先计算逻辑表达式的值，如果其值为True，则计算表达式1的值，这个值就是整个表达式的结果；否则，取表达式2的值作为整个表达式的结果。



- 例如：
- $z = x > y ? x : y ;$
- //z的值就是x、y中较大的一个值。如果 $x > y = \text{True}$ ， $z = x$ ；如果 $x > y = \text{False}$ ，则 $z = y$
- $z = x \geq 0 ? x : -x ;$       //z的值就是x的绝对值

## 3.4.2 表达式

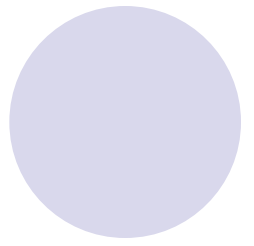
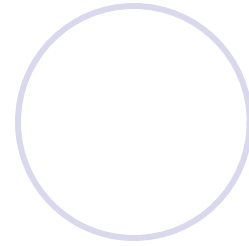
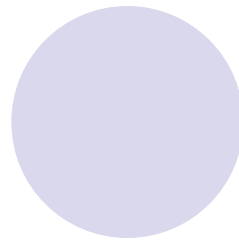
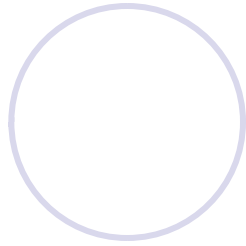
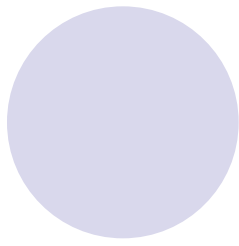
- 表达式是由操作数和运算符构成的。操作数可以是常量、变量、属性等；运算符指示对操作数进行什么样的运算。因此也可以说表达式就是利用运算符来执行某些计算并产生计算结果的语句，如`x=a+b*c`。
- 在表达式中，操作数的数据类型可以不同，只要相互兼容即可。当表达式中混合几种不同的数据类型时，**C#**会基于运算的顺序将它们自动转换成同一类型。
- 类型的自动转换是通过使用**C#**的类型提升规则来完成的。

- 如果一个操作数是**decimal**类型，另一个操作数提升为**decimal**，但**float**或**double**类型不能自动提升为**decimal**；
- 否则，如果一个操作数是**double**类型，另一个操作数提升为**double**；
- 否则，如果一个操作数是**float**类型，另一个操作数提升为**float**；
- 否则，如果一个操作数是**ulong**类型，另一个操作数提升为**ulong**，但带符号数如**sbyte**、**short**、**int**或**long**不能自动提升；
- 否则，如果一个操作数是**long**类型，另一个操作数提升为**long**；
- 否则，如果一个操作数是**uint**类型，另一个操作数若是**sbyte**、**short**或**int**，那么这两个操作数都提升为**long**；
- 否则，如果一个操作数是**uint**类型，另一个操作数提升为**uint**；
- 除以上情况之外，两个数值类型的操作数都提升为**int**类型。

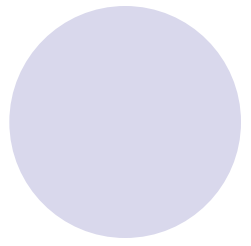
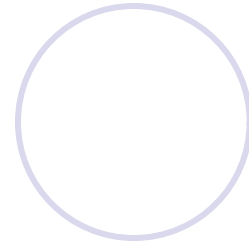
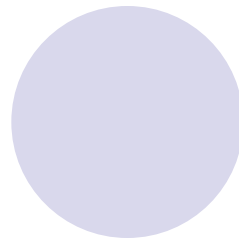
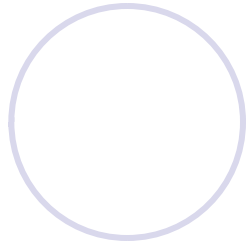
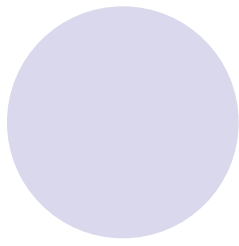


## 3.5 运算符的优先级与结合性

- 当表达式中包含多种运算符时，**C#**将按优先级从高到低的顺序和结合性规则对表达式求值 (如乘法比加法的优先级要高)。当表达式中出现两个相同优先级的运算符时，则根据结合性进行计算。左结合运算符按从左到右的顺序计算，右结合运算符按从右到左的顺序计算(如赋值运算 **$x=y=z$** )。
- 运算符按优先级从高到低的顺序如表**3-9**所示。同类的优先级相同。



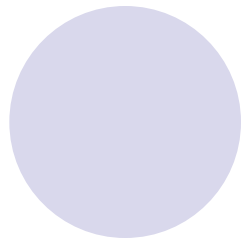
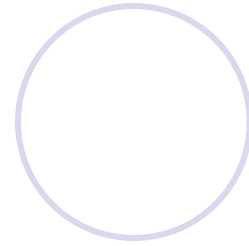
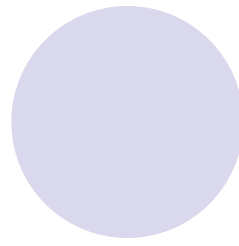
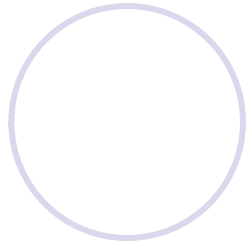
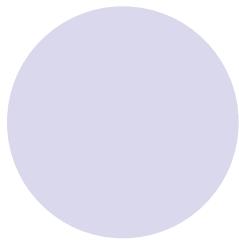
- **【例3-6】** 写出下列语句的运算结果。
- `bool a = 32 + 5 * 2 <= 6 - 3 * 2 || "abc" != "ab" && !(6 - 3 < 8 / 4);`
- 等价于：
- `bool a = ((32 + (5 * 2)) <= (6 - (3 * 2 ))) || (( "abc" != "ab" ) && (!( (6 - 3 ) < (8 / 4) )));`



- 3.6 格式控制符
- .NET有两种格式控制符，一种是标准格式控制符，另一种是用户自定义的。本节只介绍标准格式控制符
- 1. 货币金额格式
- 字符C或c用来组成将数据转换为货币金额格式的字符串。该字符后面的数字表示货币金额数据小数点后保留的数字个数。
- `int a = 12345;`
- `string str1 = String.Format("{0:c2}",a);`//输出 ¥12,345.00
- 2. 整数数据格式
- 字符D或d用来组成将数据表示为十进制整数数据的格式化字符串，其后面的数字表示数据的位数，如果这个数字小于整数数据的位数，则显示所有的整数位；如果这个数字大于整数数据的位数，则在整数数据的前面用数字“0”补足所有的位数。
- `int a = 12345;`
- `string str1 = String.Format("{0:d6}",a);`//012345

- 3. 科学计数法格式
- 字符**E**或**e**用来组成将数据转换为科学计数法形式，其后面的数字用来规定科学计数法表示数据的小数点后数字的个数。如果该字符后面没有数字，则显示7位有效数字。
- `int a = 12345;`
- `string str1 = String.Format("{0:e6}",a);`  
`//1.234500e+004`
- `string str2 = String.Format("{0:e}",a);`  
`//1.234500e+004`
- 4. 浮点数据格式
- 字符**F**或**f**用来描述带有小数点的数据的显示形式，该字符后面的数字规定了小数点后的数据位数，如果没有指定数字，则数据默认保留2位小数。如果指定数字大于数据本身小数部分的位数，则在小数部分数字的最后补“0”。
- `int a = 12345;`
- `double d = 12345.55678;`
- `string str1 = String.Format("{0:f}",a);` `//12345.00`
- `string str2 = String.Format("{0:f3}",d);` `//12345.557`

- 5. 通用数据格式
- 字符G或g用于将数据格式化为最紧凑的字符格式。该种格式符将根据具体的数据决定是用科学计数法表示，还是用定点数据格式或整数数据格式表示更紧凑，并返回更紧凑的一种格式。
- `int a = 12345;`
- `double d = 1345.55678;`
- `string str1 = String.Format("{0:g}", a);` //结果为12345
- `string str2 = String.Format("{0:g}", d);` //结果为1345.55678
- `string str3 = String.Format("{0:g4}", a);` //结果为  
`1.235e+04`
- `string str4 = String.Format("{0:g4}", d);` //结果为1346
- 其中，g4表示保留4位有效数字。
- 6. 自然数据格式
- 字符N或n用来表示自然数据格式将数据格式化成带逗号和 decimal 点的形式。
- `int a = 12345;`
- `double d = 1345.55678;`
- `string str1 = String.Format("{0:n}", a);` //结果为12,345.00
- `string str2 = String.Format("{0:n}", d);` //结果为1,345.56
- `string str3 = String.Format("{0:n3}", a);` //结果为12,345.000
- `string str4 = String.Format("{0:n3}", d);` //结果为1,345.557



- 7. 十六进制数据格式
- 字符X或x用于将数据表示为十六进制数据格式，其后面的数字表示格式化数据的数字个数，其规定与整数格式类似。
- `int a = 123455;`
- `string str1 = String.Format("{0:x}",a);` `//1e23f`
- `string str2 = String.Format("{0:x5}", a);` `//1e23f`
- `string str3 = String.Format("{0:x6}", a);` `//01e23f`
- `string str4 = String.Format("{0:x7}", a);` `//001e23f`
- 8. 保持精度(来回转换)
- 将数字从一种格式转换为另一种格式时，可能会降低精度。字符R或r用于将字符串转换为数字，保持其精度。使用该格式说明符后，运行时环境将尽可能保持原有数据的精度

## 3.7 控制台输入与输出

- 1. 控制台输入可用**Read( )**和**ReadLine( )**方法
- **Console.Read( )**作为**int**类型返回一个字符(即从标准输入流读取一个字符)，如果没有可用字符，则返回-1。如**char ch=(char)Console.Read( )**;
- 要读取一串字符，则使用**Console.ReadLine( )**方法。该方法一直读取字符，直到用户按下**Enter**键，然后将它们返回到**string**类型的对象中。
- **Console.ReadLine( )**返回一个包括下一输入行的字符串(即从标准输入流读取一行字符串)，如果没有可用行，则返回**null**。如**string s= Console.ReadLine( )**;
- 2. 控制台输出可用**Write( )**和**WriteLine( )**方法
- **Console.Write( )**输出一个或多个值，而其后不跟一个新行符。
- **Console.WriteLine( )**输出一个或多个值，但其后跟一个新行符。