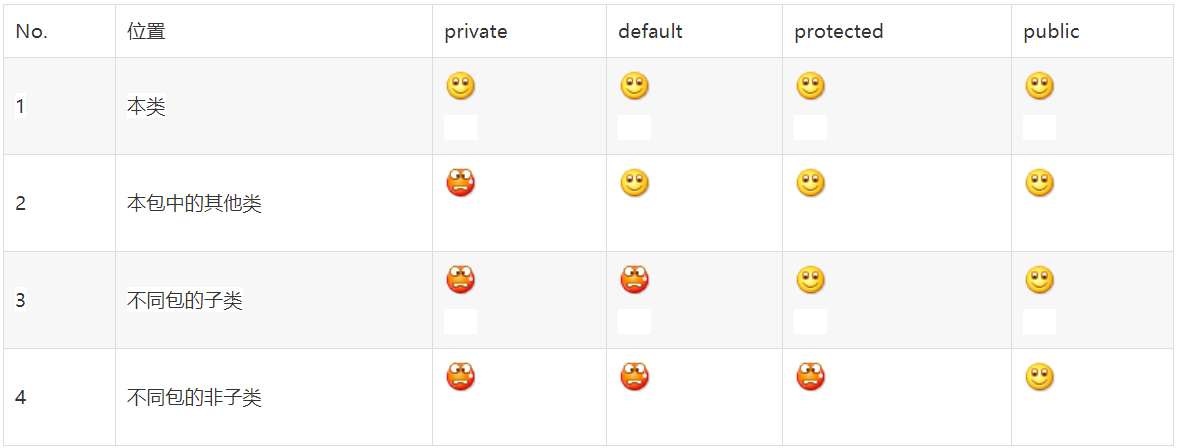
public/protected/private：访问权限



interface/abstract

abstract：用于修饰类和方法，abstract类只定义类拥有的方法，而具体的实现叫有子类实现，abstract类中即可有一般的方法也可以有abstract方法，抽象方法类中的抽象方法只声明，不能实现(注意：public abstract void test() ; 这样的才是抽象方法，而public abstract void test() { } 不能叫抽象方法)，抽象类中也可以没有抽象方法，子类在继承抽象类后，必须实现其所有的抽象方法，此时子类中的方法(从父类实现的抽象方法)就不能在用abstract修饰了，如果子类中还有抽象方法，那么子类必须也是抽象类

interface：接口解决了类单继承，接口实现了多继承，接口中只能有抽象方法，不能有实现的方法，而且接口中的方法默认的修饰符是public abstract，在接口中可以声明属性，默认的修饰符是public static final，所以接口中的属性是常量，在声明属性的时候就要为其赋值，不能是默认的值，在访问接口中的属性时，不能通过super关键字，只能通过接口.属性名，毕竟接口不是类，有抽象方法不一定是抽象类，也有可能是接口，接口中都是抽象方法

JDK1.8以后，接口中可以静态方法的实现方法

区别：

实现接口和继承抽象类的类都必须重写所有抽象方法(子类非abstract)

抽象类强调的是同一个类的共有方法，强调所属关系/接口强调的是不同类拥有相同的方法，强调的是特定功能的实现

接口中的方法只能是public的

抽象类的方法不能是private的

implements/extends

implements：实现接口

extends：继承类

continue/break

continue：退出当前循环

break：退出整个循环

instanceof：

result = object instanceof class用来判断当前对象(object)是否是类(class)的实例

System.*out*.println(person instanceof Person);

和instanceof相似的isInstance，isInstance是Class类的一个方法，也是用来判断类和对象之间的关系

System.*out*.println(Person.class.isInstance(person));

一个是关键字，一个是Class的方法

static/final

static用于修饰方法，属性，代码块

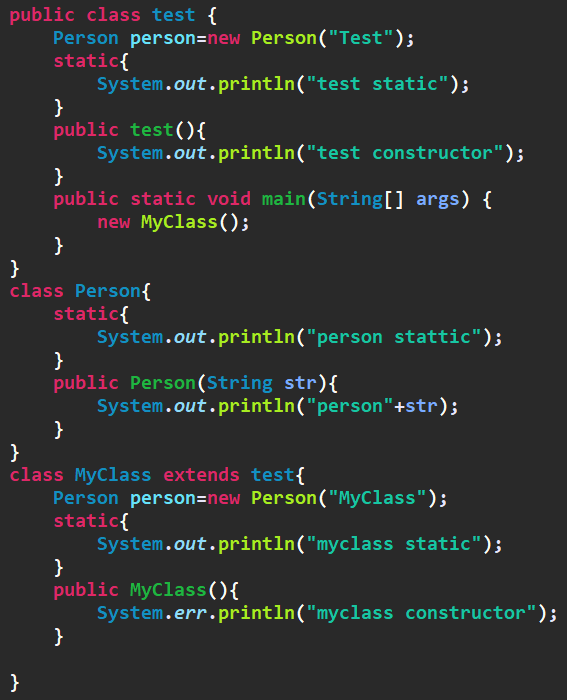
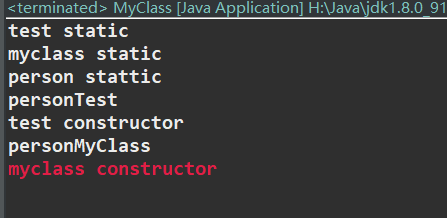
static的作用：方便在没创建对象的情况下进行调用(方法/变量)，被static关键字修饰属性和方法不需要依赖对象访问，只要类被加载即可通过类名访问

static方法：在static方法内部，不能使用this关键字和访问非静态属性/方法，因为静态属性/方法的属于类，非静态属性/方法属于对象，但是在非静态方法中可以访问静态属性和方法

static变量：静态变量(类变量)属于类，在内存只有一个，在非静态方法中可以通过this访问静态变量的(当然也可以通过类名.属性名访问)，虽然静态属性属于类，但是属性任然属于对象，只是对象之间共享这个属性，任然是可以访问的。

但是在静态方法中不能使用this，但是对象可以对static的属性进行修改

static代码块：当类加载的时候，就会执行静态代码中的内容，而且只会执行一次

首先加载main方法所在类test，test类中的static代码块会执行，所以输出test static，加载完test类后，执行new MyClass，加载MyClass类，在加载MyClass的时候，发现继承test类，然后加test类，发现test以加载，执行MyClass的static代码块myclass static，加载完MyClass类后，通过构造器生成对象(main方法中调用了new MyClass)，在生成对象之前，先要创建父类，在调用父类的构造器之前，需要初始化父类的成员变量，所以先调用new Person(“Test”)，然后加载Person类，输出静态代码块person static，然后在执行Person的构造器，输出personTest，然后创建父类，调用构造器，输出test constructor，父类创建好，创建子类MyClass，在调用构造方法之前，先初始化成员变量，new Person(“MyClass”)，Person以初始化，直接调用构造器，输出personMyClass，最后，执行new MyClass，输出myclass constructor

多个static{}的执行顺序是按出现的顺序执行

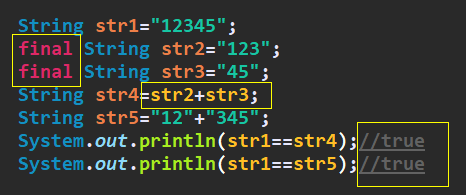
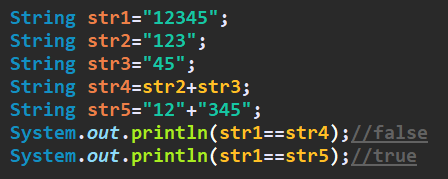
static{}/{}(构造代码块)的执行顺序：static{}是在类加载的时候执行，只执行一次，而{}每次创建对象的时候会执行，当第一次加载类的时候，先执行static，在执行{}

在static代码块中同样也可以访问权限和静态方法中一样，在静态方法中能使用的在static代码块中也可以使用，同样，在非静态方法中能使用的，在{}也可以使用，static代码的作用是在类加载前做一些初始化操作，{}代码块的作用是在创建对象的时候做一些初始化操作，构造函数的作用是创建对象，并为对象做一些操作

final：final用于修饰类，方法，属性

final修饰类时，表示此类不被继承，final修饰方法时，表示此方法不能被重写，final类中的方法实际也是final方法，既然final类都不能继承了，那么方法肯定也不能重写了

final修饰属性时表示该属性不能被修改，final修饰的属性必须为其赋初值，要么在声明的时候，要么在静态代码块中(即编译后就不能在改变)，而static修饰的属性，属于类，但是可变，有默认值，可以构造函数中声明static的属性值，但是不在构造函数中声明final的属性值，在构造函数中声明意味属性值可变。



final修饰的属性是常量，str2，str3就不可能再变了，所以str4的值也是固定的，在编译的时候就可以确定，如果不用final修饰，str4=str2+str3，str2，str3属于变量，在运行的时候能会被修改，所以，在编译的时候，不会对str2+str3进行编译，被final修饰后，就不会变，所以在编译期就可以确定

右

编译后，str4=12345(final)，运行期间在比较的是12345==12345，比较的是常量池中的地址(intern)

左

运行期间比较的是12345常量池中地址和new String(12345)[str2+str3]堆中的地址，new String创建了两个对象，

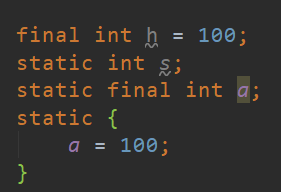
如果str1 == str4.intern，返回true

static保证内存中只有一份，而final保证不能被修改(对属性来说)

初始化常量的时候为什么必须赋初值，而没有默认值？

Static的属性有默认值，而final的属性必须在定义或构造器中赋初值，static属性虽然属于类，但是可以修改的，而final是不能修改的，所以必须提前赋初值

定义常量：private static final int i=10；final表示不可修改，既然都不可变了，每个对象都在内存中开辟空间太浪费，所以使用static修饰，只有一份，private只是对常量的可见性进行声明，所以，这里必须在定义的时候初始化



final从属于对象，final初始化在构造对象之前都可以初始化，

static从属于类，在类能被访问(类被初始化)之前都可以初始化，

final static从属于类

使用 final 方法的原是把方法锁定，以防任何继承类修改它的含义；final方法和类绑定，调用final方法比非final的更快

super/this

super主要是在子类中使用，用于访问父类的构造方法/属性和一般方法

访问构造方法：

当在new子类时，会首先去创建父类，其实在子类的构造函数中，都会隐式的添加一个super()用于访问父类的构造器(在构造器的第一行)，默认的是访问父类的无参构造器，可以在子类的构造器中显式的指定访问父类的哪个构造器super(参数1，参数2…)，但是必须在构造器中的第一行，且只能在构造器中使用

访问父类的属性和方法：

在子类的方法中访问父类属性和方法(只能在子类的方法中使用)，super.属性名/super.方法名

super表示父类

this主要用于访问当前类的属性，方法和构造器

和super一样，this可以访问当类的其他构造器，super是访问父类的构造器，this访问当前类的构造器，this(参数1，参数2….)

访问当前类的属性和方法this.属性名/方法名

并且this还可以表示当前对象，直接指向的是当前对象

this只能在非静态方法中使用

native

native用于修饰方法，native method本地方法

native关键字说明其修饰的方法是一个原生方法，方法对应实现不在当前文件中，而是在通过其他语言(如C)实现的文件中，，java本身并不能对操作系统底层访问和操作，但是可以通过JNI(java Native Interface)接口调用其它语言实现对底层的访问

当java程序需要与java外面环境交互时，比如与底层的操作系统等，可以使用本地方法，一般在java中使用native方法是通过c语言实现操作系统的底层交互



Object中方法大部分都是native的，native方法和一般方法一样，可以使用修饰符，异常机制，但是不能使用abstract修饰，虽然native方法看起来没有实现体{}，但是它调用的是其他语言的实现体，所以还是有实现体，而abstract方法是只声明，无方法体，所以不能使用abstract修饰native

synchronized/volatile

synchronized是jvm的锁，和Lock不一样，Lock是类，synchronized是关键字

volatile用于修饰属性，保证了属性的可见性和顺序性

transient

通过原生的JDK进行序列化时，transient修饰属性不会被序列化(static修饰的属性也不被序列化)，如果实现Externalizable接口，重写writeExternal和readExternal方法可以指定是否序列化transient和static的属性，Serializable接口中定义writeObject和readObject方法也可以指定是否序列化transient和static的属性

assert

断言，主要在测试类中对结果进行判断，断言

boolean/byte/char/short/int/float/long/double

boolean：1位(即只有0/1两种情况)

byte：8位，1个字节，-128~127(2的7次方，还有一位是符号位)

short：2个字节

int：4个字节

long：8个字节

long i=10；其实此时的i是int类型，当i小于32位数的变量时，都按int结果进行计算，long i=10L/long i=10l此时的i就是按long类型进行计算

float：4个字节

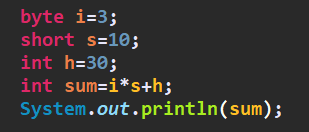
double：8个字节

浮点数默认默认类型为double，float i=10f/float i=10F

浮点数是不精确的，所以不能对浮点数进行精确运算，可以通过BigDecimal类对浮点数进行运行操作

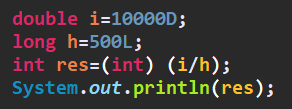
char：2个字节，java采用的是Unicode编码，一个字符占用两个字节，所以java可以表示中文

自动类型转换：byte—short—int—long—float—double



计算结果会自动往高类型转换

强制类型转换：高数据类型转向低数据类型

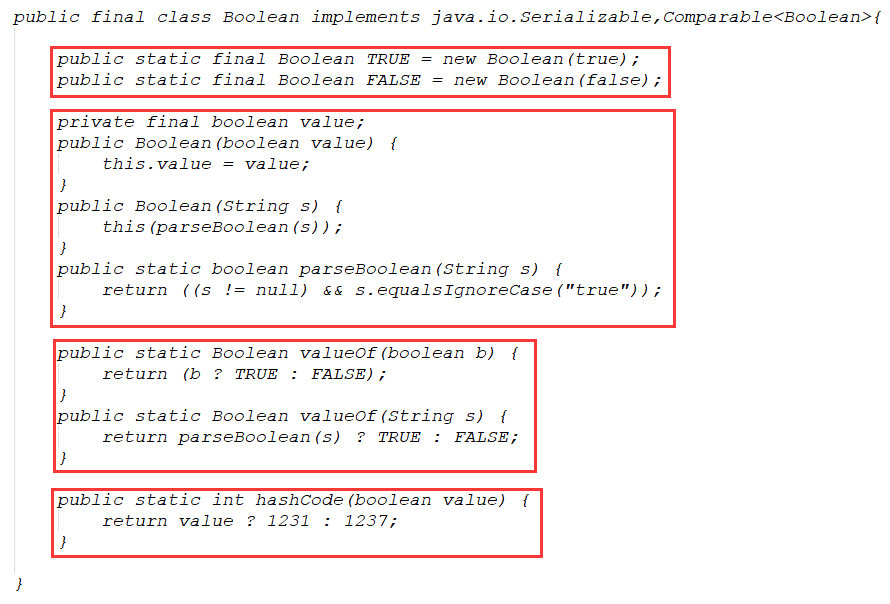


本来i/h计算结果为double(自动类型转换)，然后强制将double转换为int，强制类型转换可能会影响精度

包装类

每个基本数据类型都对应一个包装类，包装类方便对象操作，java中一切皆对象

Boolean基本类型的保证类Boolean：



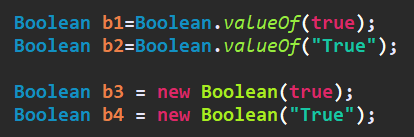
Boolean类的主要源码

Boolean是final的不可继承

public static final Boolean TRUE = new Boolean(true)：Boolean采用了享元模式，就是多个对象使用的都是同一份，例如，一个文本文件中，每个abcdefg….都是对象的话，会非常占用内存，如果将abcdef…作为共享，重复的a就可以直接引用即可，不用创建对象

为什么hashcode为素数，如果Boolean插入到Hashtable中，如果不是素数会造成hash冲突

创建Boolean的方式



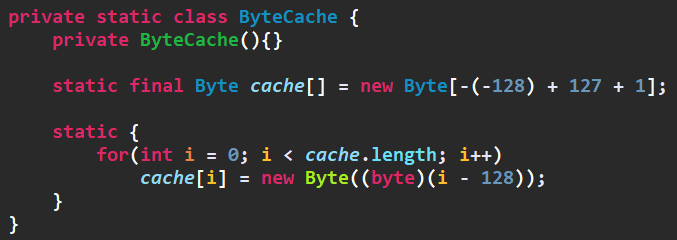
推荐使用valueOf，通过valueOf获取的Boolean是享元的，所以b1和b2是同一个对象，而new Boolean是创建一个新的Boolean，b3和b4不相等

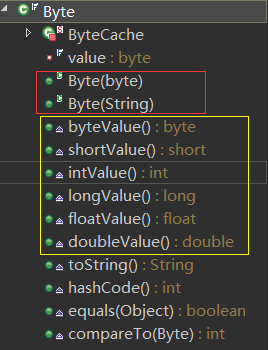
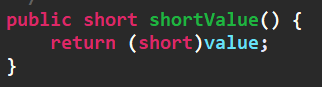
byte的包装类是Byte



Byte也是final，只能用不能继承

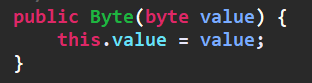
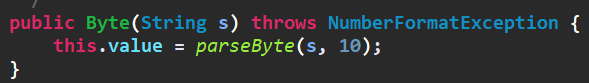
在Byte内定义了一个内部类ByteCache，用来缓存Byte



shortValue将byte强制类型转换

Byte公有两个构造器

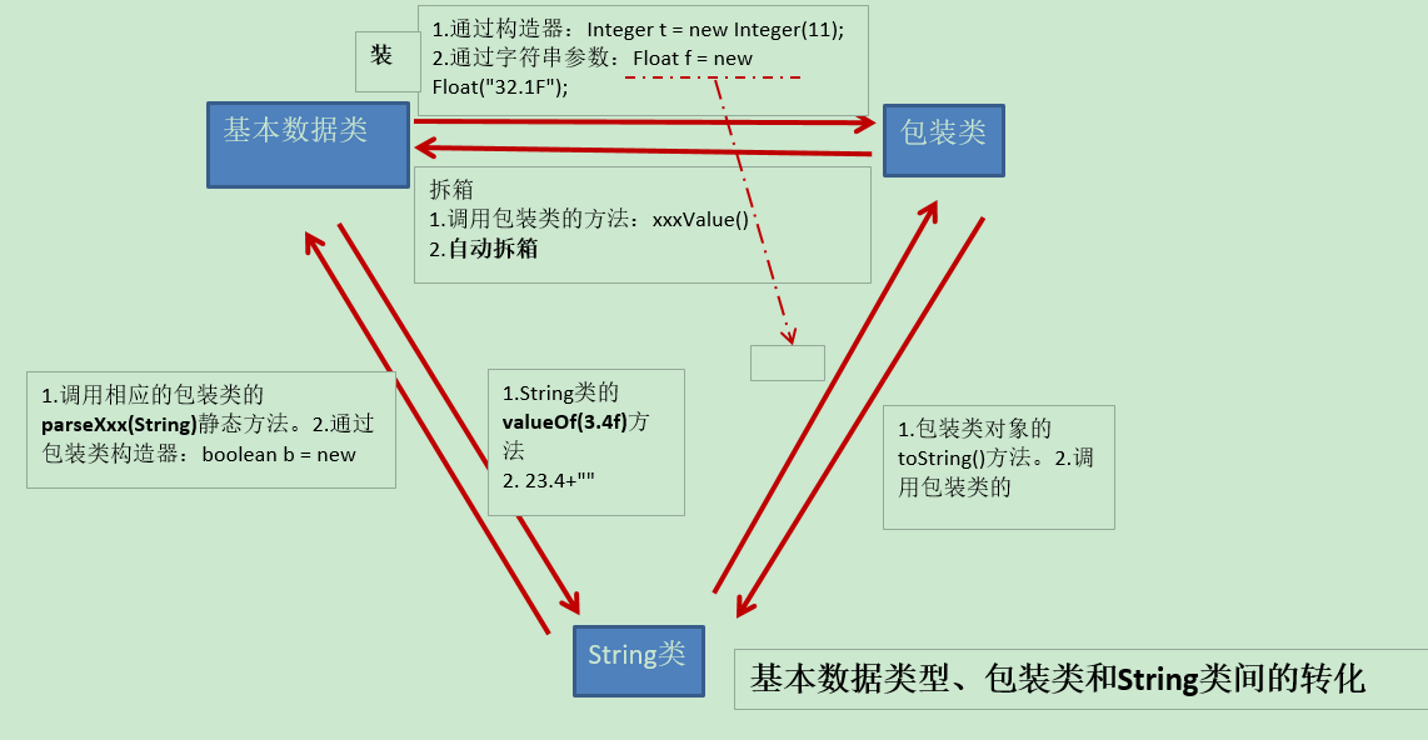
 

Byte.*parseByte*("11",10)；返回值是11----11的10进制是11

Byte.*parseByte*("11",2)；返回值是3--------3的2进制是11

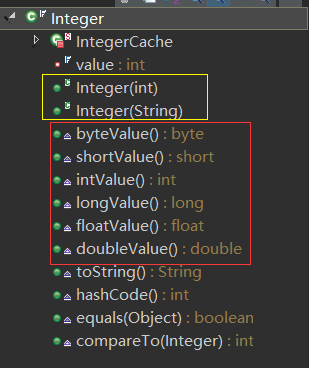
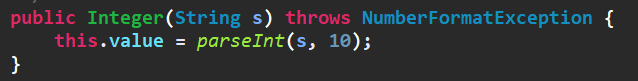
Byte.*parseByte*("11",3)；返回值是4--------4的3进制是11

parseByte就是这个意思，实际parseByte调用的是Integer的parseInt(String，int)方法



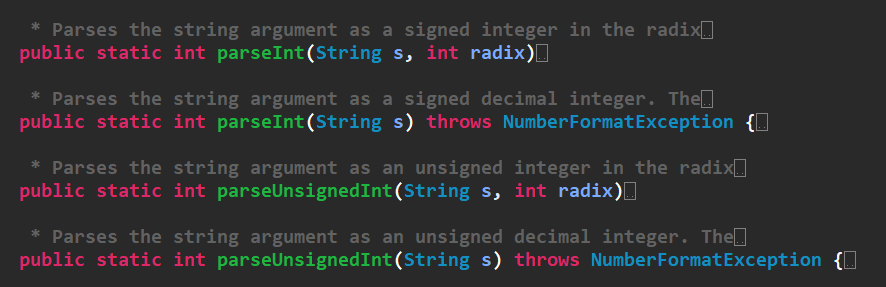
int的包装类是Integer



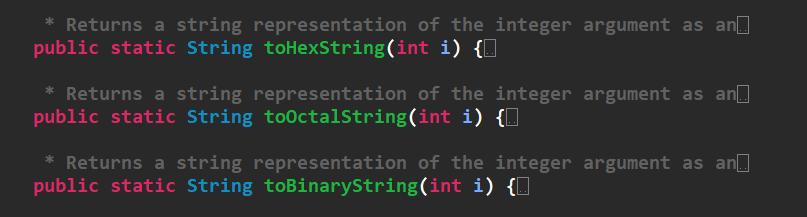
和Byte一样，Integer提供了两个构造函数，也可以将int类型向高/低数据类型转换

Integer同样也缓存了-128到127的Integer，但是并没有缓存完，比较太多了

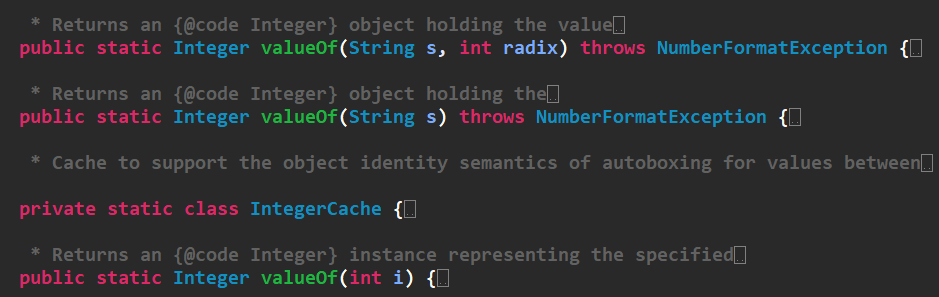


parseInt是Integer比较重要的一个方法，Byte.*parseByte*实际就是调用的这个方法，radix表示的进制

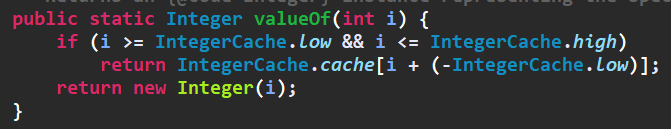
*parseUnsignedInt*的String不能是”-100”负数，parseInt可以传入负数



静态方法进行10进制数作为其他进制输出



通过静态方法获取Integer，推荐使用valueOf获取



先从IntegerCache中获取，如果没有再通过构造器创建

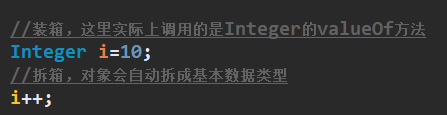
获取Integer方法共4种：构造器+valueOf(2+2)

boolean—Boolean byte---Byte short—Short int---Integer long---Long float---Float double—Double char—Character

使用方式很多，包装类提供了很多static方法，通过这些static方法可以对基本数据类型进行操作

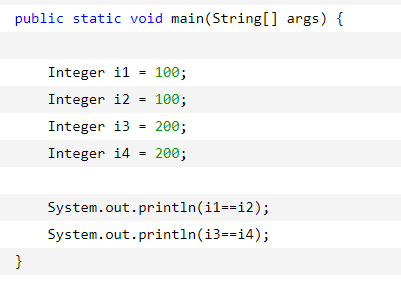
自动装箱/拆箱

自动装箱和拆箱是指基本数据类型和其对应的包装类之间可以相互转换，JDK1.5之后



在编译后的class文件中，可以找到Integer i=10；在编译后，会自动调用Integer的valueOf(int)，并返回一个Integer对象，i指向的就是这个Integer对象，拆箱会自动调用Integer的intValue方法，返回一个int类型

很多时候可以通过查看编译的class文件来了解编译的时候做了什么



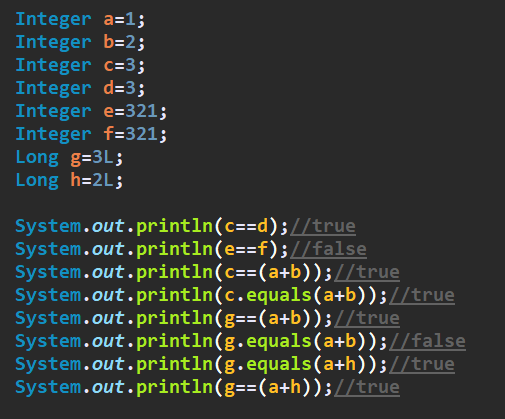
true/false

装箱后i1，i2，i3，i4指向的是Integer对象，而-128到127可以在Integer的缓存中找到，所以i1==i2是true，而200超出了缓存范围，每次都会创建新的，所以i3==i4为false

但是Double，Float没有缓存，都是直接创建

有缓存：Long，Character，Integer，Boolean，Byte，Short(缓存范围都是256，Boolean中只有两个对象，TRUE/FALSE)

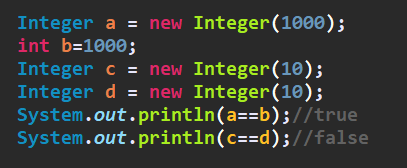
无缓存：Double，Float



c==(a+b)：a+b先拆箱再装箱，最后还是value为3的Integer对象

equals比较的是基本类型大小

g==(a+h)：a+h先拆箱进行基本运行，向上转型结果为Long，再装箱为value为3(小于127)的Long对象



通过New Integer创建的使用个新对象，而通过valueOf才是从缓存中获取，所以输出true

c==d的时候，涉及到一个拆箱的过程，所以输出true