Object类、包装类、内部类

1、本章内容

- Object类
- equals方法
- toString方法
- getClass方法
- 包装类和基本类型的转换
- 静态内部类
- 成员内部类
- 局部内部类
- 匿名内部类
- String类
- String API
- String和基本类型之间的转换包装类
- StringBuilder
- StringBuffer
- 递归

2、Object类

Object类概述

> 所在包: java.lang
>
> 类的定义: public class Object
>
> 类 Object 是类层次结构的根类。每个类都使用 Object 作为超类。所有对象（包括数组）都实现这个类的方法。
>
> 为什么要定义层次结构根类?
> Java认为所有的对象都具备一些基本的共性内容，这些内容可以不断的向上抽取，最终就抽取到了一个最顶层的类中(Object)；该类中定义的就是所有对象都具备的功能。
>
> 如果在类的声明中未使用extends关键字指明其父类，则默认父类为java.lang.Object类。

也就是说以下两种类的定义的最终效果是完全相同的：

```
class Person { }

class Person extends Object { }
```

使用Object 接收所有类的对象：

```
public class UnderStandObject {
    // 定义静态方法
    public static void fun(Object obj) {
        System.out.println(obj);
    }
```

```
    public static void main(String[] args) {
        // Person Student都是Object的子类,都可以作为参数传递进来
        fun(new Person());
        fun(new Student());
    }

}

// 人类
class Person { }

// 学生类
class Student { }
```

Object 类属于 java.lang 包，此包下的所有类在使用时无需手动导入，系统会在程序编译期间自动导入；
我们可以在JRE System Library的rt.jar里面看到常用的包以及类；

Object类结构图

> 在IDEA里面，想要查看类的层次结构,View -> Tool Windows -> Structure; 或者直接点击左下角Structure；

Object类方法详解

```
private static native void registerNatives();

public Object(){}

static {
    registerNatives();
```

```
}

public final native Class<?> getClass();

public native int hashCode();

public boolean equals(Object obj) {
    return (this == obj);
}

protected native Object clone() throws CloneNotSupportedException;

public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}

public final native void notify();

public final native void notifyAll();

public final native void wait(long timeout) throws InterruptedException;

public final void wait(long timeout, int nanos) throws InterruptedException...

public final void wait() throws InterruptedException {
    wait(0);
}

protected void finalize() throws Throwable { }
```

registerNatives方法

```
    //细心的你可能会发现，不光是Object类，甚至System类、Class类、ClassLoader类、Unsafe类等等，都能在类代码中找到如下代码：

private static native void registerNatives();

static {
    registerNatives();
}
```

本地方法的理解：

为了搞清楚这四行代码的含义和作用，我们需要先了解什么是本地方法。

本地方法在Java类中的定义是用native进行修饰，且只有方法定义，没有方法实现。

在《深入Java虚拟机》这本书的1.3.1节对Java方法有以下描述：
Java有两种方法：Java方法和本地方法。
Java方法：由Java语言编写，编译成字节码，存储在class文件中。
本地方法：使用 native 关键字说明这个方法是原生函数，也就是这个方法是用 C/C++等非 Java 语言实现的，并且被编译成了 DLL，由 java 去调用。

Java方法是平台无关的，但本地方法却不是。运行中的Java程序调用本地方法时，虚拟机装载包含这个本地方法的动态库，并调用这个方法。本地方法是联系Java程序和底层主机操作系统的连接方法。

由此可知，本地方法的实现是由其他语言编写并保存在动态连接库中，因而在java类中不需要方法实现。

为什么要用 native 方法：

1、java 使用起来非常方便，然而有些层次的任务用 java 实现起来不容易，或者我们对程序的效率很在意时，问题就来了。例如：有时 java 应用需要与 java 外面的环境交互。这是本地方法存在的主要原因，你可以想想 java 需要与一些底层系统如操作系统或某些硬件交换信息时的情况。本地方法正是这样一种交流机制：它为我们提供了一个非常简洁的接口，而且我们无需去了解 java 应用之外的繁琐的细节。

2、native 声明的方法，对于调用者，可以当做和其他 Java 方法一样使用。
一个 native method 方法可以返回任何 java 类型，包括非基本类型，而且同样可以进行异常控制。
native method 的存在并不会对其他类调用这些本地方法产生任何影响，实际上调用这些方法的其他类甚至不知道它所调用的是一个本地方法。JVM 将控制调用本地方法的所有细节。
如果一个含有本地方法的类被继承，子类会继承这个本地方法并且可以用 java语言重写这个方法（如果需要的话）。

registerNatives方法的理解：

registerNatives本质上就是一个本地方法，但这又是一个有别于一般本地方法的本地方法，从方法名我们可以猜测该方法应该是用来注册本地方法的。

上述代码的功能就是先定义了registerNatives()方法，然后当该类被加载的时候，调用该方法完成对该类中本地方法的注册。

More: Object类中的registerNatives方法的作用深入介绍
https://blog.csdn.net/Saintyyu/article/details/90452826

getClass方法

getClass方法是一个final方法，不允许子类重写，并且也是一个native方法。

返回当前运行时对象的Class对象，注意这里是运行时，比如以下代码中n是一个Number类型的实例，但是java中数值默认是Integer类型，所以getClass方法返回的是java.lang.Integer：

只要是字节码文件.class，jvm都会创建一个Class类型的对象来表示这个字节码;

```
public class GetClassTest {
    public static void main(String[] args) {
        // 返回当前运行时对象的Class对象
```

```
        System.out.println("str".getClass());// class java.lang.String

        /*
         * 每一个.class文件被jvm加载后，会在jvm中创建一个类型为Class类型的对象来表示这份字节码，
         * 你可以通过类名.class获取这个Class类型的对象，例如String.class
         */
        System.out.println("str".getClass() == String.class); // true

        // 抽象类 Number 是 BigDecimal、BigInteger、Byte、Double、Float、Integer、Long 和 Short 类的超类。
        Number n = 100;
        System.out.println(n.getClass()); // class java.lang.Integer

        Number n1 = 10.22;
        System.out.println(n1.getClass()); // class java.lang.Double

        /*
         * animal1/animal2所指向的对象和发出的方法调用在编译器是不确定的; 在运行期才能确定指向的具体实例和发出的方法调用
         * */
        Animal animal1 = new Dog();
        Animal animal2 = new Cat();
        // 如果定义了包,会包含包名
        System.out.println(animal1.getClass()); // class Dog
        System.out.println(animal2.getClass()); // class Cat
    }
}

class Animal {
}

class Dog extends Animal {
}

class Cat extends Animal {
}
```

hashCode方法

> hashCode方法也是一个native方法。
>
> 该方法返回对象的哈希码，主要使用在哈希表中，比如JDK中的HashMap。

哈希码的通用约定如下：

哈希码产生的依据：哈希码并不是完全唯一的，它是一种算法，让同一个类的对象按照自己不同的特征尽量的有不同的哈希码，但不表示不同的对象哈希码完全不同。也有相同的情况，看程序员如何写哈希码的算法。

在Java中，哈希码代表对象的特征。

哈希码百度百科：
https://baike.baidu.com/item/%E5%93%88%E5%B8%8C%E7%A0%81/5035512?fr=aladdin

哈希码的通用约定：

1、在java程序执行过程中，在一个对象没有被改变的前提下，无论这个对象被调用多少次，hashCode方法都会返回相同的整数值。对象的哈希码没有必要在不同的程序中保持相同的值。

2、如果2个对象使用equals方法进行比较并且相同的话，那么这2个对象的hashCode方法的值也必须相等。

3、如果根据equals方法，得到两个对象不相等，那么这2个对象的hashCode值不需要必须不相同。

通常情况下，不同的对象产生的哈希码是不同的。默认情况下，对象的哈希码是通过将该对象的内部地址转换成一个整数来实现的。

String的hashCode方法实现如下， 计算方法是 s[0]*31^(n-1) + s[1]*31^(n-2) + … + s[n-1]，其中s[0]表示字符串的第一个字符，n表示字符串长度：

```
private int hash; // Default to 0

// String类hashCode源码
public int hashCode() {
    int h = hash;
    if (h == 0 && value.length > 0) {
        char val[] = value;

        for (int i = 0; i < value.length; i++) {
            h = 31 * h + val[i];
        }
        hash = h;
    }
    return h;
}

public class HashCodeTest {
    public static void main(String[] args) {
        String str = "abc"; // {'a','b','c'}

        /*
         * h = 0
         * h = 31 * h + val[i];
         * h = 31 * 0 + 97  ,h = 97
         * h = 31 * 97 + 98 ,h = 3105
         * h = 31 * 3105 + 99 ,h = 96354
         * */
        int hashCode = str.hashCode();
        System.out.println(hashCode);

        /*
         * s[0]*31^(n-1) + s[1]*31^(n-2) + … + s[n-1]
         * 97 * 31^2 + 98 * 31^1 + 99
         * 93217 + 3038 + 97 = 96354
         * */
```

```
    }
}
```

equals方法

比较两个对象是否相等（其实内部比较的就是两个对象地址）。Object类的默认实现，即比较2个对象的内存地址是否相等：

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

Object类的equals方法对于任何非空引用值x和y，当x和y引用同一个对象时，此方法才返回true。这个也就是我们常说的地址相等。[所以千万不要说Java中的equals()都是来比较值的]

基本数据类型的比较用 == 如： a == 3, b == 4, a == b;(比较的是值是否相等）

引用数据直接调用 equals()方法进行比较：

```
public class Student {
    private String name;
    private int age;

    public Student() {

    }

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

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "Student [name=" + name + ", age=" + age + "]";
```

```
    }

}

public class Test_Student {
    public static void main(String[] args) {
        Student stu1 = new Student("张三", 20);
        Student stu2 = new Student("张三", 20);
        System.out.println(stu1.equals(stu2)); // false
    }
}
```

两个对象stu1 和 stu2 的内容明明相等，应该是true呀？怎么会是false?

因为此时直接调用equals()方法进行比较的是两个对象的地址，new一下，就会在堆上创建想的空间，地址自然不会相同，所以为false。

String 类对象比较使用的是 equals() 方法，实际上String 类的 equals( )方法就是覆写Object类中的 equals()方法;

那怎样比较对象的内容，才会得到true呢?

答案是： 覆写equals( )方法

还是上面的例子，覆写equals( )方法:

```
public class Student {
    private String name;
    private int age;

    public Student() {
    }

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

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
```

```
    public String toString() {
        return "Student [name=" + name + ", age=" + age + "]";
    }

    // 覆写equals方法
    public boolean equals(Object obj) {
        // 如果参数为null,直接返回false
        if (obj == null) {
            return false;
        }

        // 判断地址是否相等,地址相同就是同一个对象,内容肯定相等
        if (this == obj) {
            return true;
        }

        // 判断传入对象是否为Student类对象
        if (!(obj instanceof Student)) {
            return false;
        }

        /**
         * 传入的对象为Student类对象并且地址不相等
         * 向下转型,将Object类强转为Student类的对象,比较属性值
         */
        Student stu = (Student) obj;
        return stu.name.equals(this.name) && stu.age == this.age; // this表示当前
对象
    }

}

class Test_Student {
    public static void main(String[] args) {
        Student stu1 = new Student("张三", 20);
        Student stu2 = new Student("张三", 20);
        System.out.println(stu1.equals(stu2)); // true
    }
}
```

所以，引用类型的数据在进行比较时，应该先覆写@Override equals()方法，不然比较的这是俩个对象的内存地址，必然不会相等。

注意点：如果重写了equals方法，通常有必要重写hashCode方法，这点已经在hashCode方法中说明了。

==操作符与equals方法

```
==操作符：可以使用在基本数据类型变量和引用数据类型变量中
    基本类型比较值：只要两个变量的值相等，即为true。
        int a=5; if(a==6){…}
    引用类型比较引用(是否指向同一个对象)：只有指向同一个对象时，==才返回true。
        Person p1=new Person();
        Person p2=new Person();
        if (p1==p2){…}
    用“==”进行比较时，符号两边的数据类型必须兼容(可自动转换的基本数据类型除外)，否则编译出错
```

equals():

所有类都继承了Object，也就获得了equals()方法。还可以重写。

只能比较引用类型，其作用与“==”相同,比较是否指向同一个对象。

格式:obj1.equals(obj2)

特例：当用equals()方法进行比较时，对类File、String、Date及包装类（Wrapper Class）来说，是比较类型及内容而不考虑引用的是否是同一个对象；

原因：在这些类中重写了Object类的equals()方法。

当自定义类使用equals()时，可以重写。用于比较两个对象的“内容”是否都相等。

```
import java.util.Date;

public class EqualsTest {
    public static void main(String[] args) {
        // 基本数据类型
        int i = 10;
        int j = 10;
        double d = 10.0;
        System.out.println(i == j);//true
        System.out.println(i == d);//true

        boolean b = true;
        // System.out.println(i == b);

        char c = 10;
        System.out.println(i == c); // true

        char c1 = 'A';
        char c2 = 65;
        System.out.println(c1 == c2); // true

        // 引用类型
        Customer cust1 = new Customer("Tom", 21);
        Customer cust2 = new Customer("Tom", 21);
        System.out.println(cust1 == cust2); // false

        String str1 = new String("jeb");
        String str2 = new String("jeb");
        System.out.println(str1 == str2); // false

        System.out.println("------------------------------");
        // 重写Customer类的equals方法之后
        System.out.println(cust1.equals(cust2)); // false ---> true
        System.out.println(str1.equals(str2)); // true

        Date date1 = new Date(32432525324L);
        Date date2 = new Date(32432525324L);
        System.out.println(date1.equals(date2)); // true


    }
}

class Customer {
```

```java
    private String name;
    private int age;

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public Customer() {
        super();
    }

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

    // 自动生成的equals()
    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Customer other = (Customer) obj;
        if (age != other.age)
            return false;
        if (name == null) {
            if (other.name != null)
                return false;
        } else if (!name.equals(other.name))
            return false;
        return true;
    }


    //重写的原则：比较两个对象的实体内容(即：name和age)是否相同
    //手动实现equals()的重写
    /*@Override
    public boolean equals(Object obj) {
        if (obj == null){
            return false;
        }
```

```
        if (this == obj) {
            return true;
        }

        if (obj instanceof Customer) {
            Customer cust = (Customer) obj;
            return this.age == cust.age && this.name.equals(cust.name);
        } else {
            return false;
        }
    }*/

    //手动实现
    @Override
    public String toString() {
        return "Customer[name = " + name + ",age = " + age + "]";
    }

}
```

重写equals()方法的原则：

reflexive，自反性。任何非空引用值x，对于 x.equals(x) 必须返回true。

symmetric，对称性。任何非空引用值x和y，如果 x.equals(y) 为 true，那么 y.equals(x) 也必须为 true。

transitive，传递性。任何非空引用值x、y和z，如果 x.equals(y) 为 true 并且 y.equals(z) 为 true，那么 x.equals(z) 也必定为 true。

consistent，一致性。任何非空引用值x和y，多次调用 x.equals(y) 始终返回 true 或始终返回 false，前提是对象上 equals 比较中所用的信息没有被修改

对于任何非空引用值 x，x.equals(null) 都应返回 false。

判断题：

```
int it = 65;
float fl = 65.0f;
System.out.println("65和65.0f是否相等？" + (it == fl)); // true

char ch1 = 'A';
char ch2 = 12;
System.out.println("65和'A'是否相等？" + (it == ch1)); // true
System.out.println("12和ch2是否相等？" + (12 == ch2)); // true

String str1 = new String("hello");
String str2 = new String("hello");
System.out.println("str1和str2是否相等？"+ (str1 == str2)); // false

System.out.println("str1是否equals str2？"+(str1.equals(str2))); // true
System.out.println("hello" == new java.util.Date()); // 编译不通过
```

clone方法

创建并返回当前对象的一份拷贝。一般情况下，对于任何对象 x，表达式 x.clone() != x 为true，x.clone().getClass() == x.getClass() 也为true。

Object类的clone方法是一个protected的native方法。

protected native Object clone() throws CloneNotSupportedException；当我们在方法上看到throws某个异常；那么表示调用这个方法可能会有异常，但是没有在方法里面进行处理，谁调用那么就谁来处理;

由于Object本身没有实现Cloneable接口，所以不重写clone方法并且进行调用的话会发生CloneNotSupportedException异常。

CloneNotSupportedException异常:
　当调用 Object 类中的 clone 方法复制对象，但该对象的类无法实现 Cloneable 接口时，抛出该异常。

Cloneable接口:
　此类实现了 Cloneable 接口，以指示 Object.clone() 方法可以合法地对该类实例进行按字段复制。

```
// Object类的clone()的使用
public class CloneTest {
    public static void main(String[] args) {
        Animal a1 = new Animal("花花");
        try {
            Animal a2 = (Animal) a1.clone();
            System.out.println("原始对象: " + a1);
            a2.setName("毛毛");
            System.out.println("clone之后的对象: " + a2);
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
    }
}

class Animal implements Cloneable {
    private String name;

    public Animal() {
        super();
    }

    public Animal(String name) {
        super();
        this.name = name;
    }

    public String getName() {
        return name;
    }

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

    @Override
    public String toString() {
        return "Animal [name=" + name + "]";
```

```
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        // TODO Auto-generated method stub
        return super.clone();
    }

}
```

toString方法

toString()方法在Object类中定义，其返回值是String类型，返回类名和它的引用地址。
Object对象的默认实现，即输出类的名字@实例的哈希码的16进制：

```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}

        Copied!

import java.util.Date;

public class ToStringTest {
    public static void main(String[] args) {
        Object obj = new Object();

        // return getClass().getName() + "@" + Integer.toHexString(hashCode());
        System.out.println(obj.getClass()); // class java.lang.Object

        System.out.println(obj.getClass().getName()); // java.lang.Object

        System.out.println(obj.getClass().getSimpleName()); // Object

        System.out.println(obj.hashCode()); // 460141958

        /*
         * toHexString(int i) 以十六进制（基数 16）无符号整数形式返回一个整数参数的字符串表示形式。
         * 460141958  的十六进制  1b6d3586
         * */
        //在输出语句里面调用对象,那么就相当于对象调用了toString()
        System.out.println(obj);// java.lang.Object@1b6d3586

        // 像String、Date、File、包装类等都重写了Object类中的toString()方法。使得在调用对象的toString()时，返回"实体内容"信息
        String str = new String("MM");
        System.out.println(str); // MM

        Date date = new Date(4534534534543L);
        System.out.println(date.toString()); // Mon Sep 11 08:55:34 GMT+08:00 2113
    }
}
```

toString方法的结果应该是一个简明但易于读懂的字符串。可以根据需要在用户自定义类型中重写toString()方法，建议Object所有的子类都重写这个方法。

如String 类重写了toString()方法，返回字符串的值。

```
String s1="hello";
System.out.println(s1);//相当于System.out.println(s1.toString());
```

基本类型数据转换为String类型时，调用了对应包装类的toString()方法;

```
int a=10; System.out.println(“a=”+a);
```

我们一般情况下,对于存储数据的类去重写toString();比如:学生类、玩家类等

开发中,大部分情况下直接使用工具给我们自动重写toString()即可满足大部分需求;如果有特殊需求,那么可以自己重写里面的代码;

在进行String与其它类型数据的连接操作时，自动调用toString()方法;

```
Date now=new Date();
System.out.println("now="+now);
```

相当于

```
System.out.println("now="+now.toString());
```

面试题:

```
char[] arr = new char[]{'a', 'b', 'c'};
System.out.println(arr); // abc

int[] arr1 = new int[]{1, 2, 3};
System.out.println(arr1); // [I@1b6d3586

double[] arr2 = new double[]{1.1, 2.2, 3.3};
System.out.println(arr2); // [D@4554617c
```

notify方法

notify方法是一个native方法，并且也是final的，不允许子类重写。

唤醒一个在此对象监视器上等待的线程(监视器相当于就是锁的概念)。如果所有的线程都在此对象上等待，那么只会选择一个线程。选择是任意性的，并在对实现做出决定时发生。一个线程在对象监视器上等待可以调用wait方法。

直到当前线程放弃对象上的锁之后，被唤醒的线程才可以继续处理。被唤醒的线程将以常规方式与在该对象上主动同步的其他所有线程进行竞争。例如，唤醒的线程在作为锁定此对象的下一个线程方面没有可靠的特权或劣势。

notify方法只能被作为此对象监视器的所有者的线程来调用。一个线程要想成为对象监视器的所有者，可以使用以下3种方法:

执行对象的同步实例方法

使用synchronized内置锁

对于Class类型的对象，执行同步静态方法

一次只能有一个线程拥有对象的监视器，如果当前线程不是此对象监视器的所有者的话会抛出IllegalMonitorStateException异常

注意点:

因为notify只能在拥有对象监视器的所有者线程中调用，否则会抛出IllegalMonitorStateException异常。

notifyAll方法

跟notify一样，唯一的区别就是会唤醒在此对象监视器上等待的所有线程，而不是一个线程。

wait(long timeout)方法

wait(long timeout)方法同样是一个native方法，并且也是final的，不允许子类重写。
wait方法会让当前线程等待直到另外一个线程调用对象的notify或notifyAll方法，或者超过参数设置的timeout超时时间。
跟notify和notifyAll方法一样，当前线程必须是此对象的监视器所有者，否则还是会发生IllegalMonitorStateException异常。

wait(long timeout, int nanos) 方法

跟wait(long timeout)方法类似，多了一个nanos参数，这个参数表示额外时间（以毫微秒为单位，范围是 0-999999）。 所以超时的时间还需要加上nanos毫秒。
需要注意的是 wait(0, 0)和wait(0)效果是一样的，即一直等待。

wait() 方法

跟之前的2个wait方法一样，只不过该方法一直等待，没有超时时间这个概念。

finalize方法

finalize方法是一个protected方法，Object类的默认实现是不进行任何操作。

该方法的作用是实例被垃圾回收器回收的时候触发的操作，就好比 “死前的最后一波挣扎”。

```
public class FinalizeTest {
    public static void main(String[] args) {
        Person p = new Person("Peter", 12);
        System.out.println(p);
        p = null; // 此时对象实体就是垃圾对象，等待被回收。但时间不确定。
        System.gc(); // 强制性释放空间
    }
}

class Person {
    private String name;
    private int age;

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

    public String getName() {
        return name;
    }
```

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

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    //子类重写此方法，可在释放对象前进行某些操作
    @Override
    protected void finalize() throws Throwable {
        System.out.println("对象被释放--->" + this);
    }

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

}
```

3、包装类和基本类型转换

什么是包装类

Java中的数据类型分为基本数据类型和引用数据类型，其中基本数据类型是不具有对象特征的，也就是说它们不能像对象一样拥有属性和方法，以及对象化交互。

包装类的产生就是为了解决基本数据类型存在的这样一些问题，通过包装类可以让基本数据类型获取对象一样的特征，行使对象相关的权力。

针对八种基本数据类型定义相应的引用类型—包装类（封装类），有了类的特点，就可以调用类中的方法，Java才是真正的面向对象。

基本数据类型与包装类

```
所在包: java.lang...

类的定义:
    byte类型包装类: Byte:
        public final class Byte
            extends Number
            implements Comparable<Byte>
    boolean类型包装类: Boolean
        public final class Boolean
            extends Object
            implements Serializable, Comparable<Boolean>
    其他请见文档API:
        ...
```

特点

所有包装类都是使用 final 修饰的，不能被继承，其中数值型对应的包装类都是继承自 Number 类。

包装类是不可变类，包装类的对象被创建后，它所包含的基本类型数据就不能改变。

抽象类 Number 是 BigDecimal、BigInteger、Byte、Double、Float、Integer、Long 和 Short 类的超类。Number 的子类必须提供将表示的数值转换为 byte、double、float、int、long 和 short 的方法。

```
byte byteValue()                    以 byte 形式返回指定的数值。

abstract  double doubleValue()      以 double 形式返回指定的数值。

abstract  float floatValue()        以 float 形式返回指定的数值。

abstract  int intValue()            以 int 形式返回指定的数值。

abstract  long longValue()          以 long 形式返回指定的数值。

short shortValue()                  以 short 形式返回指定的数值。

Copied!
```

作用

作为和基本数据类型对应的类类型存在，方便涉及到对象的操作。

提供每种基本数据类型的相关属性如最大值、最小值等以及相关的操作方法。

包装类的使用

与基本数据类型的转换

装箱：基本数据类型包装成包装类的实例，分为自动装箱和手动装箱；
拆箱：获得包装类对象中包装的基本类型变量，分为自动拆箱和手动拆箱；

注：JDK1.5之后，支持自动装箱，自动拆箱。但类型必须匹配。

基本数据类型转换为包装类：

```
public class WrapperClass {
    public static void main(String[] args) {
        /*
         * 以Integer类型为例,来解释装箱和拆箱
         * 其他的都是类似的
         */
        /*
         * 一个数值10,就是基本数据类型int
         * int类型10,对应的包装类是Integer
         * 装箱: 基本数据类型包装成包装类的实例
         */
        int num = 10;
        // 基本数据类型无法调用方法
        // System.out.println(num1.toString());

        // 自动装箱
        Integer i1 = num;  // 把基本数据类型值赋值给包装类
        System.out.println(i1.toString());

        // 手动装箱
        Integer i2 = new Integer(num); // 调用构造函数,把基本数据类型转换为包装类
        System.out.println(i2.toString());
        Integer in2 = new Integer("123");
        System.out.println(in2.toString());

        //报异常
        Integer in3 = new Integer("123abc");
        System.out.println(in3.toString());

        Float f1 = new Float(12.3f);
        Float f2 = new Float("12.3");
        System.out.println(f1);
        System.out.println(f2);

        Boolean b1 = new Boolean(true);
        Boolean b2 = new Boolean("TrUe");
        System.out.println(b2);
        Boolean b3 = new Boolean("true123");
        System.out.println(b3);//false


        Order order = new Order();
        System.out.println(order.isMale);//false
        System.out.println(order.isFemale);//null

    }
}

class Order {
```

```
    boolean isMale;
    Boolean isFemale;
}
```

包装类转换为基本数据类型：

```
public class WrapperClass {
    public static void main(String[] args) {
        /*
         * 拆箱：获得包装类对象中包装的基本类型变量
         */
        Integer i2 = new Integer(100);

        // 自动拆箱
        int i3 = i2;

        // 手动拆箱
        int i4 = i2.intValue();

        Float f1 = new Float(12.3);
        float f2 = f1.floatValue();
        System.out.println(f2 + 1);
    }
}
```

与字符串的转换

```
    基本数据类型转换为字符串：
        使用String.valueOf(基本类型值)
            String fstr = String.valueOf(2.34f);

        使用包装类.toString(基本类型值)

        拼接字符串
            String intStr = 5 + "";

    字符串转换为基本数据类型、包装类：
        将该字符串封装成了包装类对象 并调用对象的方法valueOf() / 或者直接赋值给基本数据类型
变量；
            int i = new Integer("124");

        使用包装类.parseXxx(numstring); 不用建立对象 直接类名调用
```

基本数据类型、包装类转换为String类型：

```
public class WrapperClass {
    public static void main(String[] args) {
        int num1 = 10;
        //方式1：连接运算
        String str1 = num1 + "";

        //方式2：调用String的valueOf(Xxx xxx)
        float f1 = 12.3f;
        String str2 = String.valueOf(f1);
        System.out.println(str2);//"12.3"
```

```
        Double d1 = new Double(12.4);
        String str3 = String.valueOf(d1);

        System.out.println(str3);//"12.4"

        //方式3：调用String的valueOf(Xxx xxx)
        int t1 = 2;
        String t2 = Integer.toString(t1);
    }
}
```

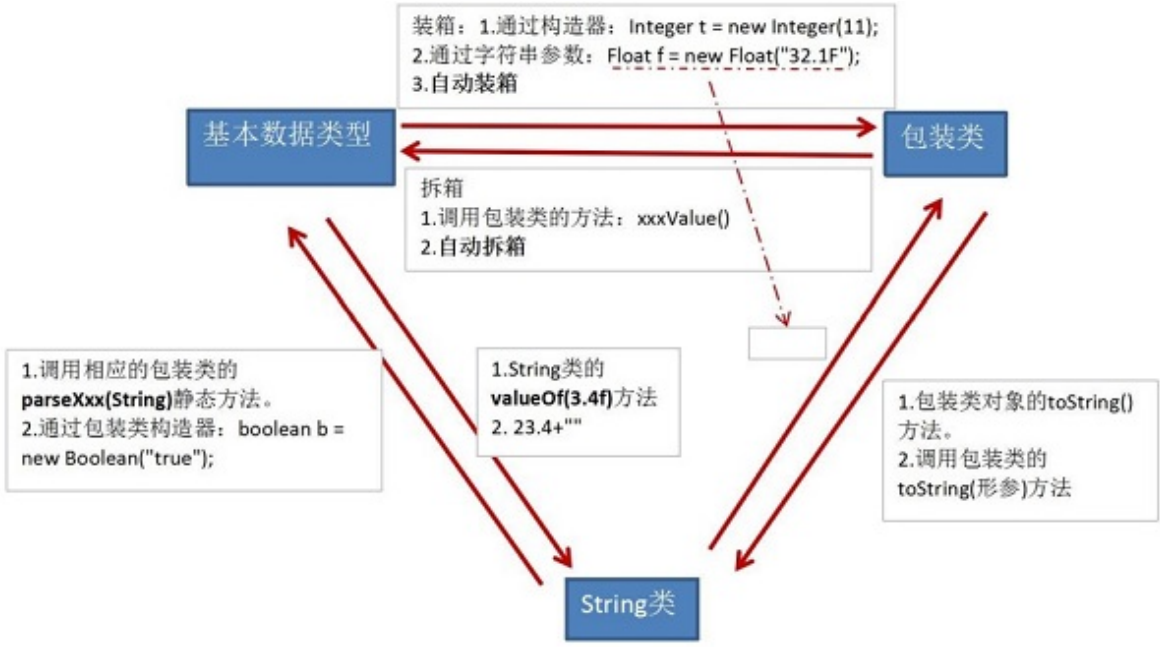
String类型转化为基本数据类型、包装类：

```
public class WrapperClass {
    public static void main(String[] args) {
        String str = "21";

        //字符串转换为基本数据类型
        //1、包装类的parseXXX()
        int t3 = Integer.parseInt(str);

        //2、包装类的valueOf() 先将字符串转换为包装类，再通过自动拆箱完成基本类型转换
        int t4 = Integer.valueOf(str);
    }
}
```

整数转成其他进制

```
    static String toBinaryString(int i) 以二进制（基数 2）无符号整数形式返回一个整数参数的字符串表示形式。

    static String toOctalString(int i)  以八进制（基数 8）无符号整数形式返回一个整数参数的字符串表示形式。

    static String toHexString(int i)    以十六进制（基数 16）无符号整数形式返回一个整数参数的字符串表示形式。

    static Integer valueOf(String s, int radix) 返回一个 Integer 对象，该对象中保存了用第二个参数提供的基数进行解析时从指定的 String 中提取的值。
```

总结：基本类型、包装类与String类间的转换



注意事项

在jdk1.5版本后，对基本数据类型对象包装类进行升级；在升级中，使用基本数据类型对象包装类可以像使用基本数据类型一样进行运算。

```
Integer i = new Integer(4); //1.5版本之前的写法
Integer i = 4; //自动装箱 1.5版本后的写法
i = i + 5;
```

i对象是不能直接和5相加的；其实底层先将i转成int类型，再和5相加，而转成int类型的操作是隐式的自动拆箱： 拆箱的原理就是i.intValue(); i+5运算完是一个int整数 如何赋值给引用类型i呢? 其实又对结果进行装箱；

```
Integer c = 127;
Integer d = 127;
System.out.println(c == d); //true
```

详情见面试题；

面试题

如下两个题目输出结果相同吗？各是什么：

```
Object o1 = true ? new Integer(1) : new Double(2.0);
System.out.println(o1);// 1.0

Object o2;
if (true){
    o2 = new Integer(1);
}else{
    o2 = new Double(2.0);
}
System.out.println(o2);// 1
```

如下代码的运行结果：

```
public class WrapperClassTest {
    public static void main(String[] args) {
        Integer i = new Integer(1);
```

```
        Integer j = new Integer(1);
        System.out.println(i == j); // false

        //Integer内部定义了IntegerCache结构，IntegerCache中定义了Integer[],保存了从-128~127范围的整数。如果我们使用自动装箱的方式，给Integer赋值的范围在-128~127范围内时，可以直接使用数组中的元素，不用再去new了。目的：提高效率
        Integer m = 1;
        Integer n = 1;
        System.out.println(m == n);// true

        Integer x = 128;//相当于new了一个Integer对象
        Integer y = 128;
        System.out.println(x == y);// false
    }
}
```

4、内部类

内部类在 Java 里面算是非常常见的一个功能了，在日常开发中我们肯定多多少少都用过，这里总结一下关于 Java 中内部类的相关知识点和一些使用内部类时需要注意的点。

从种类上说，内部类可以分为四类：成员内部类、静态内部类、匿名内部类、局部内部类。我们来一个个看：

非静态成员内部类(普通内部类/成员内部类)
静态成员内部类
局部内部类
匿名内部类---在JDK8新特性中可以使用Lambda表达式替代。(当该数据类型只使用一次/临时存在)

内部类可以访问外部类的一切资源(重要)。

成员内部类

成员内部类是最普通的内部类，它的定义位于另一个类的内部，形如下面的形式：

```
class class Circle{
    // 定义半径
    private double r;

    public Circle(){}

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

    /*
    * 定义成员内部类：位于另一个类的内部
    *   Draw类位于Circle类内部,那么Draw类就是成员内部类
    * */
    class Draw{
        public void drawCircle(){
            System.out.println("绘制圆形");
        }
    }

}
```

这样看起来，类Draw像是类Circle的一个成员，Circle称为外部类。成员内部类可以无条件访问外部类的所有成员变量、成员方法、静态变量、静态方法;

```
public class Circle {
    //成员变量
    double radius = 0;
//定义四种访问权限的成员变量
public int x = 1;
int y = 2;
protected int z = 3;
private int k = 4;

//定义一个静态变量
public static int j = 5;

//以后,只要定义了有参构造函数,那么必须把无参构造函数添加上去
public Circle() {}

//有参构造函数
public Circle(double radius) {
    this.radius = radius;
}

public void method1() {
    System.out.println("Circle的method1方法");
}

private void method2(){
    System.out.println("Circle的私有method02方法");
}

//在Circle类的内部定义了Draw
class Draw { // 内部类
    int y = 11;

    public void drawSahpe() {
        //内部类中可以访问任意的成员变量和静态变量
        System.out.println("外部类的成员变量:"+x+"\t"+z+"\t"+k);
        System.out.println("外部类的静态变量:"+j);
        System.out.println("外部类有一个成员变量y,内部类也有一个成员变量y:");
        System.out.println("内部类的y:"+this.y);
        System.out.println("外部类的y:"+Circle.this.y);

        this.method1();
        Circle.this.method1();
        method2();
        System.out.println("drawshape");
    }

    public void method1() {
        System.out.println("我是内部类的method1");
    }
}
}
```

不过要注意的是，当成员内部类拥有和外部类同名的成员变量或者方法时，会发生隐藏现象，即默认情况下访问的是成员内部类的成员。如果要访问外部类的同名成员，需要以下面的形式进行访问：

外部类.this.成员变量

外部类.this.成员方法

虽然成员内部类可以无条件地访问外部类的成员，而外部类想访问成员内部类的成员却不是这么随心所欲了。在外部类中如果要访问成员内部类的成员，必须先创建一个成员内部类的对象，再通过指向这个对象的引用来访问：

```
class Circle {
    private double radius = 0;
    public Circle(double radius) {
        this.radius = radius;
        getDrawInstance().drawSahpe();   //必须先创建成员内部类的对象，再进行访问
    }

    private Draw getDrawInstance() {
        return new Draw();
    }

    class Draw {     //内部类
        public void drawSahpe() {
            System.out.println(radius);  //外部类的private成员
        }
    }
}
```

成员内部类是依附外部类而存在的，也就是说，如果要创建成员内部类的对象，前提是必须存在一个外部类的对象。创建成员内部类对象的一般方式如下：

```
 public class Test {
    public static void main(String[] args)  {
        //第一种方式：
        Outter outter = new Outter();
        Outter.Inner inner = outter.new Inner();  //必须通过Outter对象来创建
 //第二种方式：
    Outter.Inner inner1 = outter.getInnerInstance();
}
     }

class Outter {
    private Inner inner = null;
    public Outter() {
        }

public Inner getInnerInstance() {
    if(inner == null)
        inner = new Inner();
    return inner;
}

class Inner {
    public Inner() {
         System.out.println("创建成员内部类对象 - Inner");
    }
}
```

```
}
```

内部类可以拥有 private 访问权限、protected 访问权限、public 访问权限及包访问权限。比如上面的例子，如果成员内部类 Inner 用 private 修饰，则只能在外部类的内部访问，如果用 public 修饰，则任何地方都能访问；如果用 protected 修饰，则只能在同一个包下或者继承外部类的情况下访问；如果是默认访问权限，则只能在同一个包下访问。这一点和外部类有一点不一样，外部类只能被 public 和包访问两种权限修饰。我个人是这么理解的，由于成员内部类看起来像是外部类的一个成员，所以可以像类的成员一样拥有多种权限修饰。

```
总结:
    成员内部类可以使用private、default、protected、public任意进行修饰。
    成员内部类必须寄存在一个外部类对象里。因此，如果有一个成员内部类对象那么一定存在对应的外部类对象。成员内部类对象单独属于外部类的某个对象。
    成员内部类可以直接访问外部类的成员，但是外部类不能直接访问成员内部类成员。
    成员内部类不能有静态方法、静态属性和静态初始化块。
    外部类的静态方法、静态代码块不能访问成员内部类，包括不能使用成员内部类定义变量、创建实例。
```

静态内部类

静态内部类也是定义在另一个类里面的类，只不过在类的前面多了一个关键字static。

静态内部类是不需要依赖于外部类对象的，这点和类的静态成员属性有点类似，并且它不能使用外部类的非static成员变量或者方法。

```
public class Test {
    public static void main(String[] args)  {
        Outter.Inner inner = new Outter.Inner();
    }
}

class Outter {
    public Outter() {

    }

    static class Inner {
        public Inner() {
            System.out.println("创建静态内部类对象 - Inner");
        }
    }
}
```

```
class Outter {
    int  a = 10;
    static int b = 5;
    public Outter() {

    }

    static class Inner {
        public Inner() {
            System.out.println(a);
            System.out.println(b);
        }
    }
}
```

```
public class Outter {
    private int x = 1;
    int y = 2;
    protected int z = 3;
    public int k = 4;
    static int j = 5;

    public Outter() {

    }

    public void outterMethod() {
        System.out.println("外部类的成员方法");
    }

    public static void outterStaticMethod() {
        System.out.println("外部类的静态方法");
    }

    public static Inner getInnerInstance() {
        return new Inner();
    }

    static class Inner {
        // String name;
        // static int age;

        // static {}
        // public static void gg() {}

        public Inner() {
            System.out.println("创建静态内部类对象 - Inner");
        }
        //Cannot make a static reference to the non-static field x
        public void innerMethod() {
            //所有的外部类成员变量都不可访问,因为静态内部类不依赖于外部类对象
            //System.out.println("外部类的成员变量:"+x+"\t"+y+"\t"+z+"\t"+k+"\t");
            System.out.println("外部类的静态变量:"+j);
```

```
            //不能访问外部类的成员方法
            //outterMethod();

            outterStaticMethod();
        }
    }

}


public class Test_Outter {
    public static void main(String[] args) {
        //静态内部类的创建不依赖于外部类
        //1、通过外部类实例化静态内部类对象
        //在这里实际上是new的  Outter里面的静态类
        Outter.Inner inner = new Outter.Inner();
        inner.innerMethod();

        //2、通过调用外部类的静态方法
        Inner innerInstance = Outter.getInnerInstance();
        innerInstance.innerMethod();
    }
}
```

总结:

静态内部类也是定义在另一个类里面的类，只不过在类的前面多了一个关键字static。

静态内部类不依赖于外部类对象。

静态内部类可以访问外部类的静态变量和静态方法;但是对于非静态成员变量和成员方法不能访问;

静态内部类可以定义成员变量、成员方法甚至于静态变量、静态方法和静态代码块;

局部内部类

局部内部类是定义在一个方法或者一个作用域里面的类，它和成员内部类的区别在于局部内部类的访问仅限于方法内或者该作用域内。

```
/*
 * 不是一个外部类,只是给内部类的父类
 */
public class People {
    public People() {
        System.out.println("Person的无参构造函数");
    }
    public void method() {

    }
    public static void main(String[] args) {
        Man man = new Man();

        //上溯造型:父类有的可以调用,子类扩展的不能调用
        People woman = man.getWoman();

        /*
         * 怎么调用woman里面的方法呢?
```

```
         *  1.强转,但是Woman虽说是People的子类,但是只能在方法里面看到这个类,JVM找不到这个类,失败
         *  2.让getWoman这个方法返回一个Woman类型,但是Woman类型在方法内部可见,不能作为返回值类型,失败
         *  3.最终,让People类也定义method方法,那么子类的就可以被调用到了
         */

        woman.method();
    }
}

class Man{
    private int x = 1;
    static int j = 5;

    public Man(){
        System.out.println("Man的无参构造函数");
    }

    //Man的成员方法
    public People getWoman(){
        //在成员方法里面定义一个内部类,那么这个内部类我们称之为局部内部类
        class Woman extends People{
            int age =0;

            public void method() {
                System.out.println("外部类的成员变量:"+x);
                System.out.println("外部类的静态变量:"+j);
            }
        }
        return new Woman();
    }
}
```

注意: 局部内部类就像是方法里面的一个局部变量一样，是不能有 public、protected、private 以及 static 修饰符的。

匿名内部类

匿名内部类有多种形式，其中最常见的一种形式莫过于在方法参数中新建一个接口对象 / 类对象，并且实现这个接口声明 / 类中原有的方法了：

```
    public class InnerClassTest {
    //各种修饰符的成员变量
    public int field1 = 1;
    protected int field2 = 2;
    int field3 = 3;
    private int field4 = 4;
    static int x = 10;

    //无参构造函数
    public InnerClassTest() {
        System.out.println("创建 InnerClassTest 对象");
    }

    // 自定义接口
```

```
    interface OnClickListener {
        void onClick(Object obj);
    }

    private void anonymousClassTest() {
        // 在这个过程中会新建一个匿名内部类对象，
        // 这个匿名内部类实现了 OnClickListener 接口并重写 onClick 方法
        /*
         * 当我们去new 接口/类(),在后面加上语句块,其实相当于创建了一个匿名内部类对象
         * 并且这个你们内部类对象是实现了该接口,并且要重写接口里面的方法
         * clickListener就是一个匿名内部类对象
         */
        OnClickListener clickListener = new OnClickListener() {
            // 可以在内部类中定义属性，但是只能在当前内部类中使用，
            // 无法在外部类中使用，因为外部类无法获取当前匿名内部类的类名，
            // 也就无法创建匿名内部类的对象
            int field = 1;

            @Override
            public void onClick(Object obj) {
                System.out.println("对象 " + obj + " 被点击");
                System.out.println("其外部类的 field1 字段的值为: " + field1);
                System.out.println("其外部类的 field2 字段的值为: " + field2);
                System.out.println("其外部类的 field3 字段的值为: " + field3);
                System.out.println("其外部类的 field4 字段的值为: " + field4);
            }

        };
        // new Object() 过程会新建一个匿名内部类，继承于 Object 类，
        // 并重写了 toString() 方法
        clickListener.onClick(new Object() {
            @Override
            public String toString() {
                return "obj1";
            }
        });
    }

    public static void main(String[] args) {
        InnerClassTest outObj = new InnerClassTest();
        outObj.anonymousClassTest();
    }
}
```

运行结果：

```
创建 InnerClassTest 对象
对象 obj1 被点击
其外部类的 field1 字段的值为: 1
其外部类的 field2 字段的值为: 2
其外部类的 field3 字段的值为: 3
其外部类的 field4 字段的值为: 4
```

上面的代码中展示了常见的两种使用匿名内部类的情况：
1、直接 new 一个接口，并实现这个接口声明的方法，在这个过程其实会创建一个匿名内部类实现这个接口，并重写接口声明的方法，然后再创建一个这个匿名内部类的对象并赋值给前面的 OnClickListener 类型的引用；
2、new 一个已经存在的类 / 抽象类，并且选择性的实现这个类中的一个或者多个非 final 的方法，这个过程会创建一个匿名内部类对象继承对应的类 / 抽象类，并且重写对应的方法。

同样的，在匿名内部类中可以使用外部类的属性，但是外部类却不能使用匿名内部类中定义的属性，因为是匿名内部类，因此在外部类中无法获取这个类的类名，也就无法得到属性信息。

内部类作用

1. 内部类可以很好的实现隐藏，一般的非内部类，是不允许有 private 与protected权限的，但内部类可以

2. 内部类拥有外围类的所有元素的访问权限

3. 可以实现多重继承

4. 可以避免修改接口而实现同一个类中两种同名方法的调用。

作用一:实现隐藏

平时我们对类的访问权限，都是通过类前面的访问修饰符来限制的，一般的非内部类，是不允许有 private 与protected权限的，但内部类可以，所以我们能通过内部类来隐藏我们的信息。可以看下面的例子 **接口:**

```
public interface InterfaceTest {
    void increment();
}
```

具体类:

```
public class Example {
    //使用private修饰内部类
    private class InsideClass implements InterfaceTest {
        public void test() {
            System.out.println("这是一个测试");
        }
    }

    public InterfaceTest getIn() {
        return new InsideClass();
    }
}
```

调用测试:

```
public class TestExample {
    public static void main(String args[]) {
        Example a=new Example();
        InterfaceTest a1=a.getIn();
        a1.test();
    }
}
```

从这段代码里面我只知道Example的getIn()方法能返回一个InterfaceTest 实例但我并不知道这个实例是这么实现的。而且由于InsideClass 是private的，所以我们如果不看代码的话根本看不到这个具体类的名字，所以说它可以很好的实现隐藏。

作用二:可以无条件地访问外围类的所有元素

```
public class TagBean {
    private String name="Jimbo";

    private class InTest {
        public InTest() {
            System.out.println(name);
        }
    }

    public void test()
        new InTest();
     }

    public static void main(String args[]) {
        TagBean bb=new TagBean();
        bb.test();
    }
}
```

看上面加粗部分，name这个变量是在TagBean里面定义的私有变量。这个变量在内部类中可以无条件地访问System.out.println(name);

作用三:可以实现多重继承

这个特点非常重要，个人认为它是内部类存在的最大理由之一。正是由于他的存在使得Java的继承机制更加完善。大家都知道Java只能继承一个类，它的多重继承在我们没有学习内部类之前是用接口来实现的。但使用接口有时候有很多不方便的地方。比如我们实现一个接口就必须实现它里面的所有方法。而有了内部类就不一样了。它可以使我们的类继承多个具体类或抽象类。大家看下面的例子。

类一:

```
public class Example1 {
    public String name() {
        return "liutao";
    }
}
```

类二:

```
public class Example2 {
    public int age() {
        return 25;
    }
}
```

类三:

```
public class MainExample {
    private class test1 extends Example1 {
        public String name() {
            return super.name();
        }
     }

     private class test2 extends Example2 {
         public int age() {
          return super.age();
         }
     }

    public String name() {
        return new test1().name();
    }

    public int age() {
        return new test2().age();
    }

    public static void main(String args[]) {

        MainExample mi=new MainExample();

        System.out.println("姓名:"+mi.name());

        System.out.println("年龄:"+mi.age());

    }
}
```

大家注意看类三，里面分别实现了两个内部类 test1,和test2 ，test1类又继承了Example1，test2继承了Example2，这样我们的类三MainExample就拥有了Example1和Example2的方法和属性，也就间接地实现了多继承。

作用四:避免修改接口而实现同一个类中两种同名方法的调用

大家假想一下如果，你的类要继承一个类，还要实现一个接口，可是你发觉你继承的类和接口里面有两个同名的方法怎么办？你怎么区分它们？？这就需要我们的内部类了。看下面的代码

接口:

```
public interface Incrementable {
    void increment();
}
```

类 MyIncrement

```
public class MyIncrement {
    public void increment() {

        System.out.println("Other increment()");

    }

    static void f(MyIncrement f)  {

        f.increment();

    }
}
```

大家看上面，两个方法都是一样的。在看下面这个类要继承这个类和实现这个接口；

如果不用内部类：

```
public class Callee2 extends MyIncrement implements Incrementable {

    public void increment() {

        //代码

    }
}
```

想问一下大家increment()这个方法是属于覆盖MyIncrement这里的方法呢？还是Incrementable这里的方法。我怎么能调到MyIncrement这里的方法？显然这是不好区分的。而我们如果用内部类就很好解决这一问题了。看下面代码:

```
public class Callee2 extends MyIncrement {

    private int i=0;

    private void incr() {
        i++;
        System.out.println(i);
    }

    private class Closure implements Incrementable {
        public void increment() {
            incr();
        }
    }

    Incrementable getCallbackReference() {
        return new Closure();
    }
}
```

我们可以用内部类来实现接口，这样就不会与外围类的方法冲突了

5、String 字符串类

String类概述

所在包: java.lang

类的定义:

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence
```

String 类代表字符串。Java 程序中的所有字符串字面值（如 "abc" ）都作为此类的实例实现。String是一个final类，代表不可变的字符序列。

字符串是常量，用双引号引起来表示；它们的值在创建之后不能更改。字符串缓冲区支持可变的字符串。因为 String 对象是不可变的，所以可以共享。

String对象的字符内容是存储在一个字符数组value[]中的。

public interface Serializable

类通过实现 java.io.Serializable 接口以启用其序列化功能。未实现此接口的类将无法使其任何状态序列化或反序列化。序列化接口没有方法或字段，仅用于标识可序列化的语义。

public interface CharSequence

CharSequence 是 char 值的一个可读序列。此接口对许多不同种类的 char 序列提供统一的只读访问。

已知实现类: CharBuffer, Segment, String, StringBuffer, StringBuilder

它里面有四个方法的定义:

char charAt(int index) 返回指定索引的 char 值。

int length() 返回此字符序列的长度。

CharSequence subSequence(int start, int end) 返回一个新的CharSequence，它是此序列的子序列。

String toString() 返回一个包含此序列中字符的字符串，该字符串与此序列的顺序相同。

public interface Comparable<T>

此接口强行对实现它的每个类的对象进行整体排序。这种排序被称为类的自然排序，类的compareTo 方法被称为它的自然比较方法。

它里面定义了一个方法:

int compareTo(T o) 比较此对象与指定对象的顺序。

String类结构图

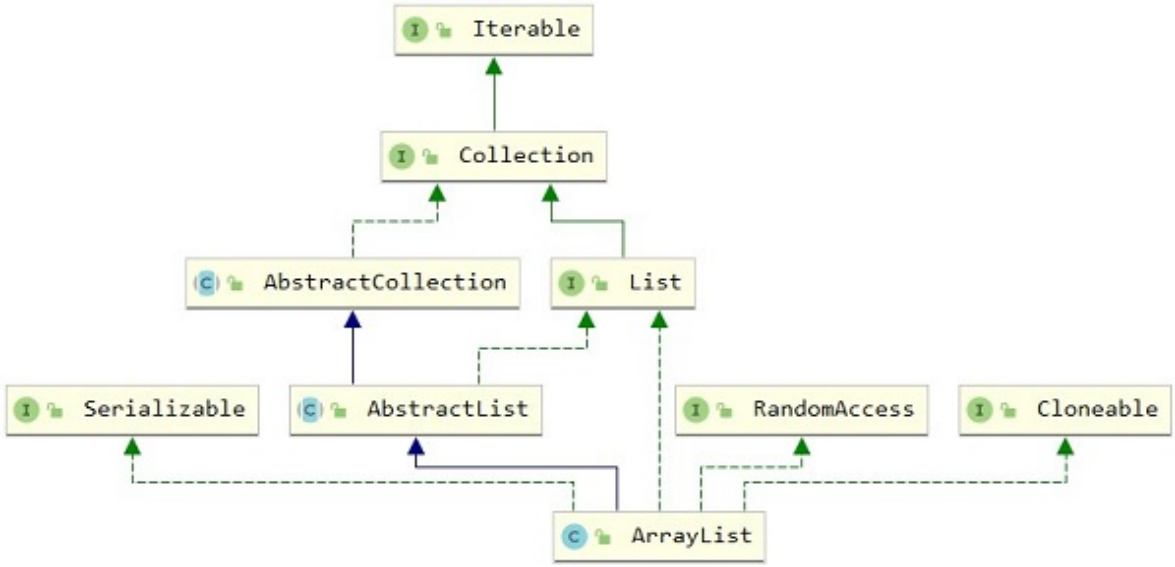
IDEA查看类结构图

以ArrayList为例，来展示类的示意图、类的层次结构、类的成员；

类的示意图:

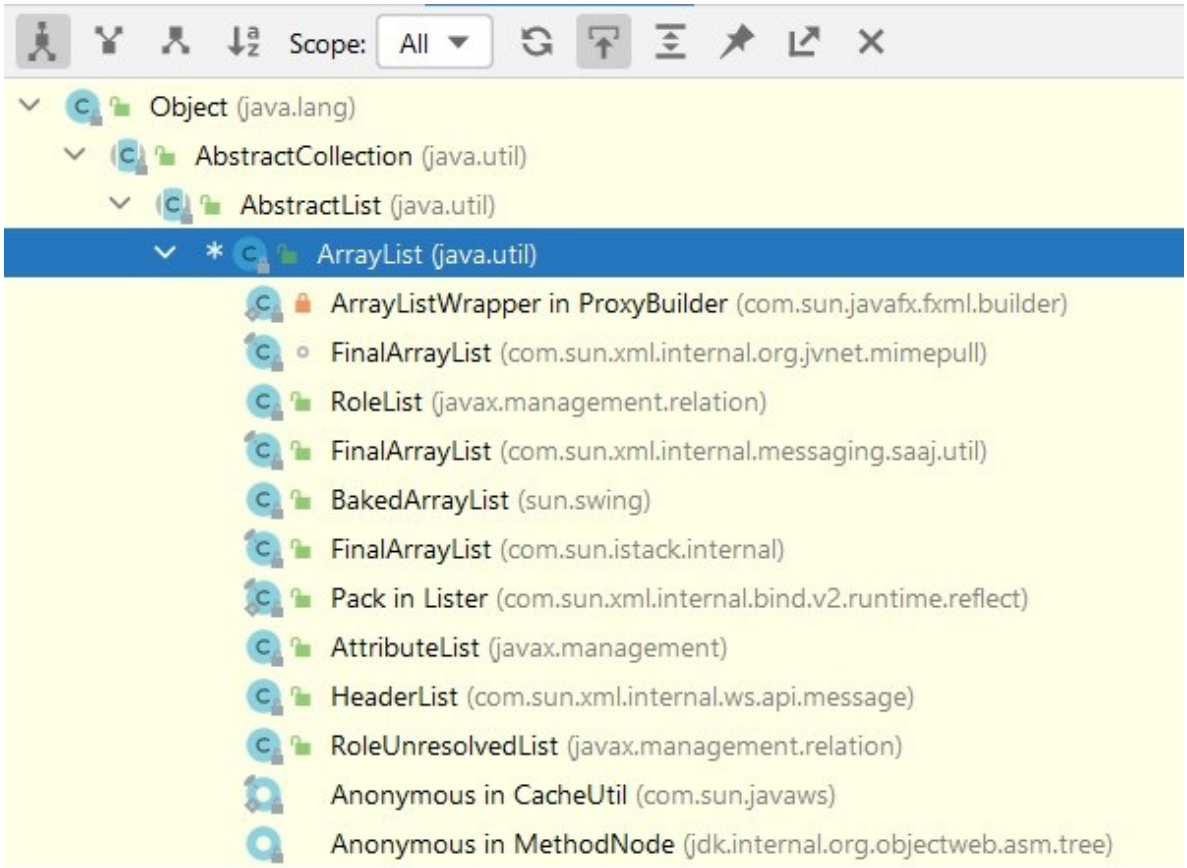
在指定的类中，右键 -> Diagrams -> Show Diagrams 或 Show Diagrams Popup;
快捷键： Ctrl + Alt + Shift + U 或 Ctrl + Alt + U;

经过这样的操作后，我们就可以看到类、接口的继承关系非常清晰地呈现在了我们眼前；同时，这个关系图还可以用各种方式来呈现，方法是在空白处右键 -> Layout -> 选择布局。



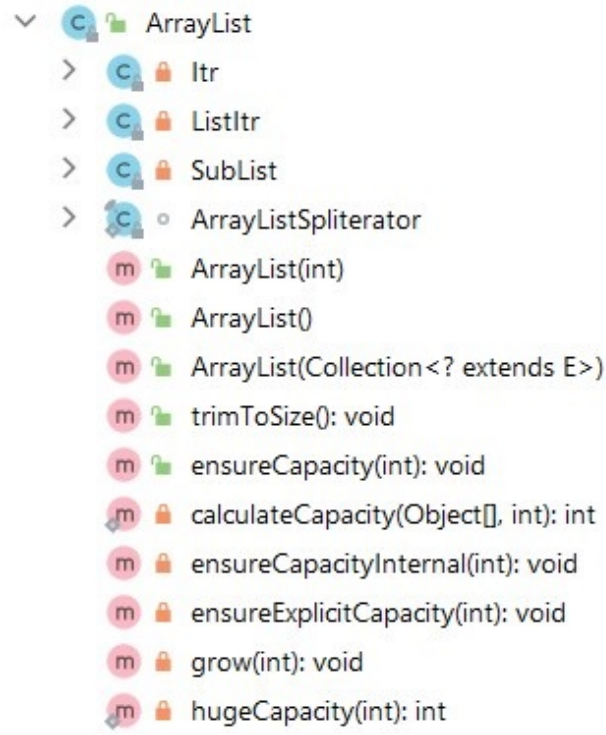
类的层次结构：

```
在指定的类中，菜单Navigate -> Type Hierarchy；
快捷键： Ctrl + H；

经过这样的操作后，我们就可以看到当前类的父类、接口、子类等；
    如果只是想看父类、接口，可以点击Supertypes Hierarchy；
    如果只是想看子类，可以点击Subtypes Hierarchy；
    如果排序查看，可以点击Sort Alphabetically；
```



类的成员：

```
在指定的类中，菜单View -> Tool Windows -> Structure;
快捷键： 左下角Structure；
```


ArrayList
Itr
ListItr
SubList
ArrayListSpliterator
ArrayList(int)
ArrayList()
ArrayList(Collection<? extends E>)
trimToSize(): void
ensureCapacity(int): void
calculateCapacity(Object[], int): int
ensureCapacityInternal(int): void
ensureExplicitCapacity(int): void
grow(int): void
hugeCapacity(int): int

String类结构图

方法摘要

构造

`String()` 初始化一个新创建的 `String` 对象，使其表示一个空字符序列。

`String(String original)` 初始化一个新创建的 `String` 对象，使其表示一个与参数相同的字符序列；换句话说，新创建的字符串是该参数字符串的副本。

`String(byte[] bytes)` 通过使用平台的默认字符集解码指定的 `byte` 数组，构造一个新的 `String`。

`String(byte[] bytes, int offset, int length)` 通过使用平台的默认字符集解码指定的 `byte` 子数组，构造一个新的 `String`。

```java
public class StringConstructorTest {
    public static void main(String[] args) {
        // String() 初始化一个新创建的 String 对象，使其表示一个空字符序列。
        String str1 = new String(); // "" 空字符串  " " 空格字符串
        System.out.println("str1的值:" + str1); // str1的值:

        // String(String original)      初始化一个新创建的 String 对象，使其表示一个与参数相同的字符序列；换句话说，新创建的字符串是该参数字符串的副本。
        String str2 = new String("123"); // String str2 = "123";
        System.out.println("str2的值:" + str2); // str2的值:123

        // 参数为byte数组相关的构造函数,非常重要;我们在之后的IO中会用到
        // String(byte[] bytes) 通过使用平台的默认字符集解码指定的 byte 数组，构造一个新的 String。
        byte b[] = {97, 98, 99, 100, 101, 102}; // a,b,c,d
        String str3 = new String(b);
        System.out.println("str3的值:" + str3); // str3的值:abcdef

        // String(byte[] bytes, int offset, int length) 通过使用平台的默认字符集解码指定的 byte 子数组，构造一个新的 String。
        String str4 = new String(b, 1, 3);
        System.out.println("str4的值:" + str4); // str4的值:bcd
    }
}
```

String(byte[] bytes, Charset charset) 通过使用指定的 charset 解码指定的 byte 数组，构造一个新的 String。

String(byte[] bytes, String charsetName) 通过使用指定的 charset 解码指定的 byte 数组，构造一个新的 String。（使用这种更加简洁）

String(byte[] bytes, int offset, int length, Charset charset) 通过使用指定的 charset 解码指定的 byte 子数组，构造一个新的 String。

String(byte[] bytes, int offset, int length, String charsetName) 通过使用指定的字符集解码指定的 byte 子数组，构造一个新的 String。

```java
import java.io.UnsupportedEncodingException;

public class StringConstructorTest {
    public static void main(String[] args) throws UnsupportedEncodingException {
        /*
         * 字节数
         * 中文：ISO:1  GBK：2   UTF-8:3
         * 数字或字母：ISO:1   GBK:1   UTF-8:1
         * */
        String username = "中";

        /*
         * byte[] getBytes(String charsetName) 使用指定的字符集将此 String 编码为 byte 序列，并将结果存储到一个新的 byte 数组中。
         *
         * String(byte[] bytes, Charset charset)  通过使用指定的 charset 解码指定的 byte 数组，构造一个新的 String。
```

```
         * String(byte[] bytes, String charsetName) 通过使用指定的 charset 解码指定的 byte 数组，构造一个新的 String。（使用这种更加简洁）
         * */
        byte[] u_iso = username.getBytes("ISO-8859-1");
        byte[] u_gbk = username.getBytes("GBK");
        byte[] u_utf8 = username.getBytes("UTF-8");
        System.out.println(u_iso.length); // 1
        System.out.println(u_gbk.length); // 2
        System.out.println(u_utf8.length); // 3

        // 跟上面刚好是逆向的，字节数组---->字符串
        String un_iso = new String(u_iso, "ISO-8859-1");
        String un_gbk = new String(u_gbk, "GBK");
        String un_utf8 = new String(u_utf8, "UTF-8");
        System.out.println(un_iso); // ?
        System.out.println(un_gbk); // 中
        System.out.println(un_utf8); // 中

        // 有时候必须是iso字符编码类型，那处理方式如下
        String un_utf8_iso = new String(u_utf8, "ISO8859-1");
        System.out.println("utf-8数组通过ISO8859-1解析成字符串:" + un_utf8_iso);

        // 将iso编码的字符串进行还原
        String un_iso_utf8 = new String(un_utf8_iso.getBytes("ISO8859-1"), "UTF-8");
        System.out.println(un_iso_utf8);
    }
}
```

String(char[] value)　　分配一个新的 String，使其表示字符数组参数中当前包含的字符序列。

String(char[] value, int offset, int count)　　分配一个新的 String，它包含取自字符数组参数一个子数组的字符。

String(int[] codePoints, int offset, int count)　　分配一个新的 String，它包含 Unicode 代码点数组参数一个子数组的字符。

String(StringBuffer buffer)　　分配一个新的字符串，它包含字符串缓冲区参数中当前包含的字符序列。

String(StringBuilder builder)　　分配一个新的字符串，它包含字符串生成器参数中当前包含的字符序列。

//总结：除了传递数组[byte/char数组]的构造函数之外,其他的构造函数基本上可以不使用,而且也最好不要使用;

```
public class StringConstructorTest {
    public static void main(String[] args) {
        // String(char[] value) 分配一个新的 String，使其表示字符数组参数中当前包含的字符序列。
        char[] chars = new char[]{'a', 'b', 'c', 'd'};
        String str1 = new String(chars);
        System.out.println("str1的值:" + str1); // str1的值:abcd

        // String(char[] value, int offset, int count)　　分配一个新的 String，它包含取自字符数组参数一个子数组的字符。
        String str2 = new String(chars, 1, 2);
```

```
        System.out.println("str2的值:" + str2); // str2的值:bc

        // String(int[] codePoints, int offset, int count)      分配一个新的String，它包含 Unicode 代码点数组参数一个子数组的字符。
        int[] ints = new int[]{97, 98, 99, 100};
        String str3 = new String(ints, 1, 2);
        System.out.println("str3的值:" + str3); // str3的值:bc

        // String(StringBuffer buffer)      分配一个新的字符串，它包含字符串缓冲区参数中当前包含的字符序列。
        StringBuffer stringBuffer = new StringBuffer("abcd");
        String str4 = new String(stringBuffer);
        System.out.println("str4的值:" + str4); // str4的值:abcd
    }
}
```

成员方法

获取

```
    char charAt(int index)  返回指定索引处的 char 值。

    int codePointAt(int index)  返回指定索引处的字符（Unicode 代码点）。

    int codePointBefore(int index)  返回指定索引之前的字符（Unicode 代码点）。

    int codePointCount(int beginIndex, int endIndex)    返回此 String 的指定文本范围中的 Unicode 代码点数。

    int hashCode()  返回此字符串的哈希码。

    int indexOf(int ch)     返回指定字符在此字符串中第一次出现处的索引。

    int indexOf(int ch, int fromIndex)  返回在此字符串中第一次出现指定字符处的索引，从指定的索引开始搜索。

    int indexOf(String str)     返回指定子字符串在此字符串中第一次出现处的索引。

    int indexOf(String str, int fromIndex)  返回指定子字符串在此字符串中第一次出现处的索引，从指定的索引开始。

    String intern()     返回字符串对象的规范化表示形式。

    int lastIndexOf(int ch)     返回指定字符在此字符串中最后一次出现处的索引。

    int lastIndexOf(int ch, int fromIndex)  返回指定字符在此字符串中最后一次出现处的索引，从指定的索引处开始进行反向搜索。

    int lastIndexOf(String str) 返回指定子字符串在此字符串中最右边出现处的索引。

    int lastIndexOf(String str, int fromIndex)  返回指定子字符串在此字符串中最后一次出现处的索引，从指定的索引开始反向搜索。

    int length()    返回此字符串的长度。

    String substring(int beginIndex)    返回一个新的字符串，它是此字符串的一个子字符串。
```

String substring(int beginIndex, int endIndex) 返回一个新字符串，它是此字符串的一个子字符串。

判断

boolean contains(CharSequence s) 当且仅当此字符串包含指定的 char 值序列时，返回 true。

boolean contentEquals(CharSequence cs) 将此字符串与指定的 CharSequence 比较。

boolean contentEquals(StringBuffer sb) 将此字符串与指定的 StringBuffer 比较。

boolean endsWith(String suffix) 测试此字符串是否以指定的后缀结束。

boolean equals(Object anObject) 将此字符串与指定的对象比较。

boolean equalsIgnoreCase(String anotherString) 将此 String 与另一个 String 比较，不考虑大小写。

boolean isEmpty() 当且仅当 length() 为 0 时返回 true。

boolean matches(String regex) 告知此字符串是否匹配给定的正则表达式。 正则部分再讲解。

boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len) 测试两个字符串区域是否相等。

boolean startsWith(String prefix) 测试此字符串是否以指定的前缀开始。

boolean startsWith(String prefix, int toffset) 测试此字符串从指定索引开始的子字符串是否以指定前缀开始。

equals 与 contentEquals 的异同：

String中的equals与contentEquals，这2个方法都可以用来比较String对象内容是否相同。

但是equals只能对2个String对象内容比较，否则返回false；

contentEquals比较类型为java.lang.CharSequence的对象内容是否相同。

有contentEquals(CharSequence cs) 为什么还需要定义contentEquals(StringBuffer sb)：

我们通过底层源码可以看出来，contentEquals(StringBuffer sb) 调用的就是contentEquals(CharSequence cs)，至于为什么要这么定义，因为一个是1.4版本提供的，一个是1.5版本提供的。

```
public boolean contentEquals(StringBuffer sb) {
    return contentEquals((CharSequence)sb);
}
```

转换

String concat(String str)　将指定字符串连接到此字符串的结尾。

static String copyValueOf(char[] data)　返回指定数组中表示该字符序列的 String。

static String copyValueOf(char[] data, int offset, int count)　返回指定数组中表示该字符序列的 String。

byte[] getBytes()　使用平台的默认字符集将此 String 编码为 byte 序列，并将结果存储到一个新的 byte 数组中。

byte[] getBytes(Charset charset)　使用给定的 charset 将此 String 编码到 byte 序列，并将结果存储到新的 byte 数组。

byte[] getBytes(String charsetName)　使用指定的字符集将此 String 编码为 byte 序列，并将结果存储到一个新的 byte 数组中。

void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)　将字符从此字符串复制到目标字符数组。

String replace(char oldChar, char newChar)　返回一个新的字符串，它是通过用 newChar 替换此字符串中出现的所有 oldChar 得到的。

String replace(CharSequence target, CharSequence replacement)　使用指定的字面值替换序列替换此字符串所有匹配字面值目标序列的子字符串。

String replaceAll(String regex, String replacement) 使用给定的 replacement 替换此字符串所有匹配给定的正则表达式的子字符串。

String replaceFirst(String regex, String replacement)　使用给定的 replacement 替换此字符串匹配给定的正则表达式的第一个子字符串。

String[] split(String regex)　根据给定正则表达式的匹配拆分此字符串。

String[] split(String regex, int limit)　根据匹配给定的正则表达式来拆分此字符串。

char[] toCharArray()　将此字符串转换为一个新的字符数组。

String toLowerCase()　使用默认语言环境的规则将此 String 中的所有字符都转换为小写。

String toUpperCase()　使用默认语言环境的规则将此 String 中的所有字符都转换为大写。

String toString()　返回此对象本身（它已经是一个字符串！）。

String trim()　返回字符串的副本，忽略前导空白和尾部空白。

static String valueOf(char c)　返回 char 参数的字符串表示形式。

static String valueOf(char[] data, int offset, int count)　返回 char 数组参数的特定子数组的字符串表示形式。

static String valueOf(boolean b)　返回 boolean 参数的字符串表示形式。

static String valueOf(char c)　返回 char 参数的字符串表示形式。

static String valueOf(double d)　返回 double 参数的字符串表示形式。

static String valueOf(float f)　返回 float 参数的字符串表示形式。

static String valueOf(int i)　　返回 int 参数的字符串表示形式。

static String valueOf(long l)　　返回 long 参数的字符串表示形式。

static String valueOf(Object obj)　　返回 Object 参数的字符串表示形式。

static String format(Locale l, String format, Object... args)　　使用指定的语言环境、格式字符串和参数返回一个格式化字符串。

static String format(String format, Object... args) 使用指定的格式字符串和参数返回一个格式化字符串。

String.format()的详细用法:
https://blog.csdn.net/anita9999/article/details/82346552

replace 和 replaceAll 的区别以及用法:

replace和replaceAll是JAVA中常用的替换字符的方法

区别:

1、replace的参数是char 和 CharSequence，即可以支持字符的替换，也支持字符串的替换(CharSequence 即字符串序列的意思,说白了也就是字符串);

2、replaceAll的参数是regex，即基于正则表达式的替换，比如，可以通过replaceAll("\\d", "*")把一个字符串所有的数字字符都换成星号;

相同点:

都是全部替换，即把源字符串中的某一字符或字符串全部换成指定的字符或字符串，如果只想替换第一次出现的，可以使用replaceFirst()，这个方法也是基于规则表达式的替换，但与replaceAll()不同的是，只替换第一次出现的字符串;

另外，如果replaceAll()和replaceFirst()所用的参数据不是基于规则表达式的，则与replace()替换字符串的效果是一样的，即这两者也支持字符串的操作;

还有一点注意:：执行了替换操作后,源字符串的内容是没有发生改变的。

```
public class StringMethodTest {
    public static void main(String[] args) {
        String src = new String("ab43a2c43d");
        System.out.println(src.replace("3", "f")); // ab4fa2c4fd
        System.out.println(src.replaceAll("\\d", "f")); // abffafcffd
        System.out.println(src.replaceAll("a", "f")); // fb43f2c43d
        System.out.println(src.replaceFirst("\\d", "f")); // abf3a2c43d
        System.out.println(src.replaceFirst("4", "h")); // abh3a2c43d

        String str = "12hello34world5java7891mysql456";
        //把字符串中的数字替换成,, 如果结果中开头和结尾有，的话去掉
        String string = str.replaceAll("\\d+", ",").replaceAll("^,|,$", "");
        System.out.println(string);

        String str1 = "12345";
        //判断str字符串中是否全部有数字组成，即有1-n个数字组成
        boolean matches = str1.matches("\\d+");
        System.out.println(matches);

        String tel = "0571-4534289";
        //判断这是否是一个杭州的固定电话
```

```
        boolean result = tel.matches("0571-\\d{7,8}");
        System.out.println(result);
    }
}
```

split结合正则：

```
public class StringMethodTest {
    public static void main(String[] args) {
        String str = "hello|world|java";
        String[] strs = str.split("\\|");
        for (int i = 0; i < strs.length; i++) {
            System.out.println(strs[i]);
        }
        System.out.println();
        String str2 = "hello.world.java";
        String[] strs2 = str2.split("\\.");
        for (int i = 0; i < strs2.length; i++) {
            System.out.println(strs2[i]);
        }
    }
}
```

比较

```
int compareTo(String anotherString) 按字典顺序比较两个字符串。

int compareToIgnoreCase(String str) 按字典顺序比较两个字符串，不考虑大小写。

两个方法的返回值是整型，按字典顺序比较两个字符串。如果第一个字符和参数的第一个字符不等，结束比较,返回他们之间的差值，如果第一个字符和参数的第一个字符相等，则以第二个字符和参数的第二个字符做比较，以此类推，直至比较的字符或被比较的字符有一方全比较完，这时就比较字符的长度.

该比较基于字符串中各个字符的 Unicode 值。按字典顺序将此 String 对象表示的字符序列与参数字符串所表示的字符序列进行比较。如果按字典顺序此 String 对象位于参数字符串之前，则比较结果为一个负整数。如果按字典顺序此 String 对象位于参数字符串之后，则比较结果为一个正整数。如果这两个字符串相等，则结果为 0；compareTo 只在方法 equals(Object) 返回 true 时才返回 0。
```

```
public class StringMethodTest {
    public static void main(String[] args) {
        String s1 = "abc";
        String s2 = "abcd";
        String s3 = "abcdfg";
        String s4 = "1bcdfg";
        String s5 = "cdfg";
        System.out.println(s1.compareTo(s2)); // -1 (前面相等,s1长度小1)
        System.out.println(s1.compareTo(s3)); // -3 (前面相等,s1长度小3)
        System.out.println(s1.compareTo(s4)); // 48（"a"的ASCII码是97,"1"的的ASCII码是49,所以返回48)
        System.out.println(s1.compareTo(s5)); // -2 ("a"的ASCII码是97,"c"的ASCII码是99,所以返回-2)
    }
}
```

字符串内存简析

字符串字面量内存存储简析

```
String s1 = "abc";//字面量的定义方式
String s2 = "abc";
s1 = "hello";
```

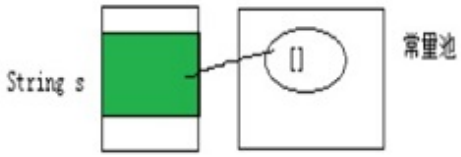
构造器创建字符串内存存储简析

String s;

未初始化

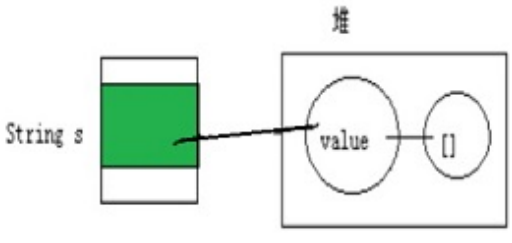
String s = null;

初始化为null

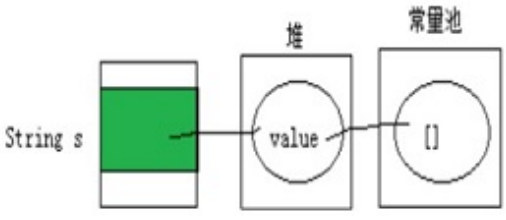


String s = "";

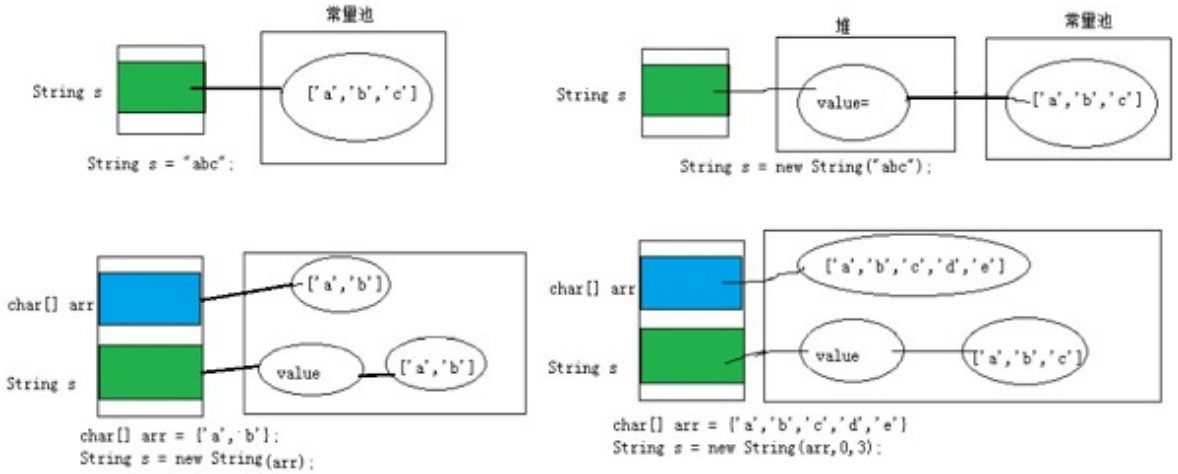
初始化为空字符串常量



String s = new String();



String s = new String("");



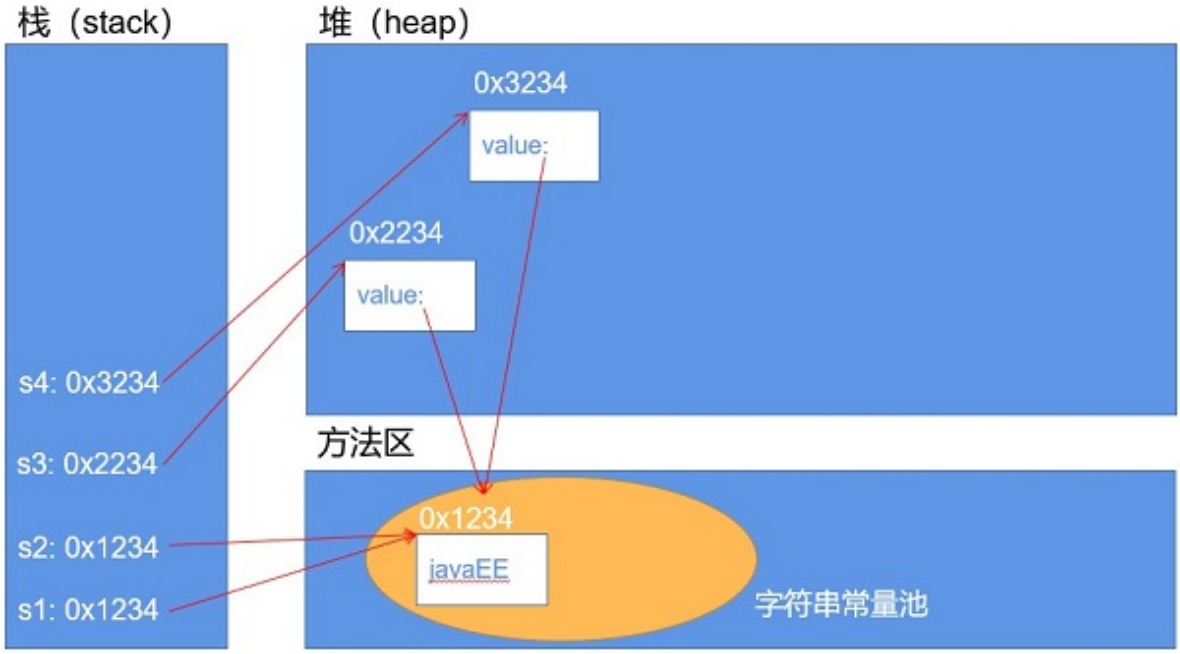
对比字面量和构造器内存存储

String str1 = “abc”;与String str2 = new String(“abc”);的区别？
字符串常量存储在字符串常量池，目的是共享；字符串非常量对象存储在堆中。

练习题分析内存存储

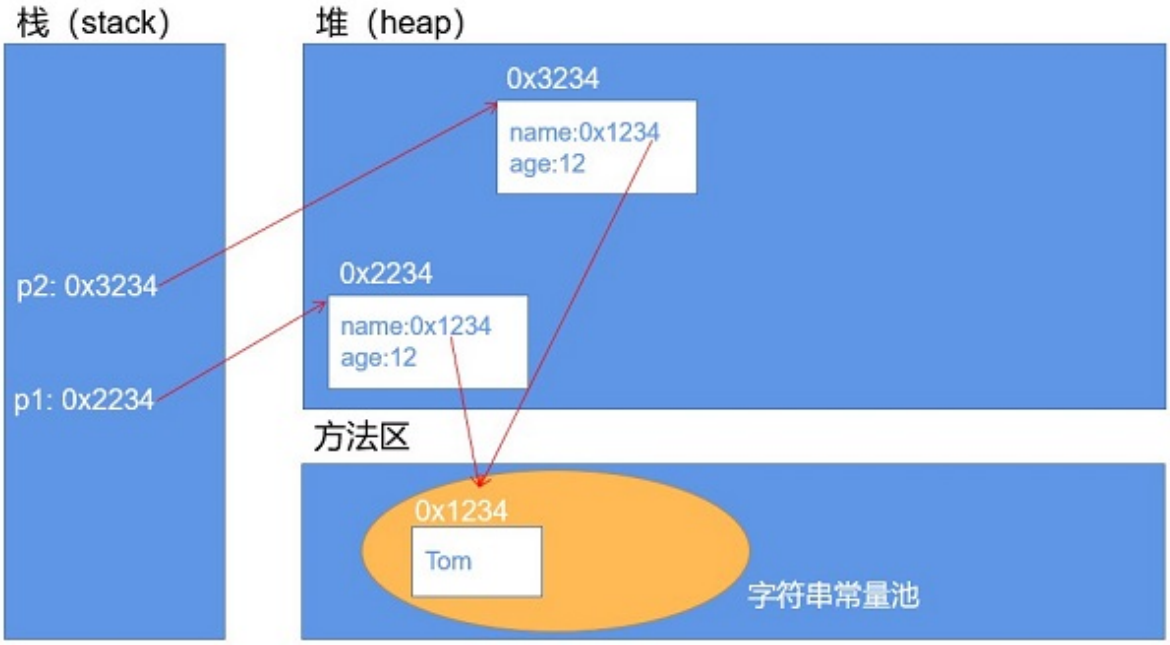
练习题一：

```
String s1 = "javaEE";
String s2 = "javaEE";
String s3 = new String("javaEE");
String s4 = new String("javaEE");
System.out.println(s1 == s2);//true
System.out.println(s1 == s3);//false
System.out.println(s1 == s4);//false
System.out.println(s3 == s4);//false
```



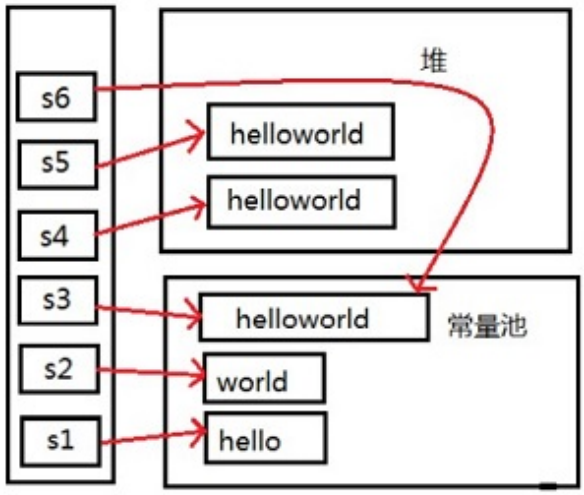
练习题二：

```
Person p1 = new Person();
p1.name = "jimbo";
Person p2 = new Person();
p2.name = "jimbo";
System.out.println(p1.name.equals(p2.name)); // true
System.out.println(p1.name == p2.name); // true
System.out.println(p1.name == "jimbo"); // true

String s1 = new String("bcde");
String s2 = new String("bcde");
System.out.println(s1 == s2); // false


Person p1 = new Person("Tom",12);
Person p2 = new Person("Tom",12);
System.out.println(p1.name == p2.name);//true
```



练习题三：

```
String s1 = "hello";
String s2 = "world";
String s3 = "hello" + "world";
String s4 = s1 + "world";
String s5 = s1 + s2;
String s6 = (s1 + s2).intern();
System.out.println(s3==s4);//false
System.out.println(s3==s5);//false
System.out.println(s4==s5);//false
System.out.println(s3==s6);//true
```



结论：

常量与常量的拼接结果在常量池。且常量池中不会存在相同内容的常量。

只要其中有一个是变量，结果就在堆中。

如果拼接的结果调用intern()方法，返回值就在常量池中。

String使用陷阱：

String s1 = "a";

说明：在字符串常量池中创建了一个字面量为"a"的字符串。

s1 = s1 + "b";

说明：实际上原来的“a”字符串对象已经丢弃了，现在在堆空间中产生了一个字符串s1+"b"（也就是"ab"）。如果多次执行这些改变串内容的操作，会导致大量副本字符串对象存留在内存中，降低效率。如果这样的操作放到循环中，会极大影响程序的性能。

String s2 = "ab";

说明：直接在字符串常量池中创建一个字面量为"ab"的字符串。

String s3 = "a" + "b";

说明：s3指向字符串常量池中已经创建的"ab"的字符串。

String s4 = s1.intern();

说明：堆空间的s1对象在调用intern()之后，会将常量池中已经存在的"ab"字符串赋值给s4。

练习题四：

下列程序运行的结果：

```
public class StringTest {
    String str = new String("good");
    char[] ch = { 't', 'e', 's', 't' };
    public void change(String str, char ch[]) {
        str = "test ok";
        ch[0] = 'b';
    }
    public static void main(String[] args) {
        StringTest ex = new StringTest();
        ex.change(ex.str, ex.ch);
        System.out.println(ex.str);// good
        System.out.println(ex.ch); // best
```

```
    }
}
```

字符串深入理解

String有两种赋值方式，第一种是通过“字面量”赋值。什么是字符串字面量？一个字符串字面量就是两个双引号之间的字符序列，形如“string”、“literal”。

```
String str = "Hello";
```

第二种是通过new关键字创建新对象。

```
String str = new String("Hello");
```

要弄清楚这两种方式的区别，首先要知道他们在内存中的存储位置。

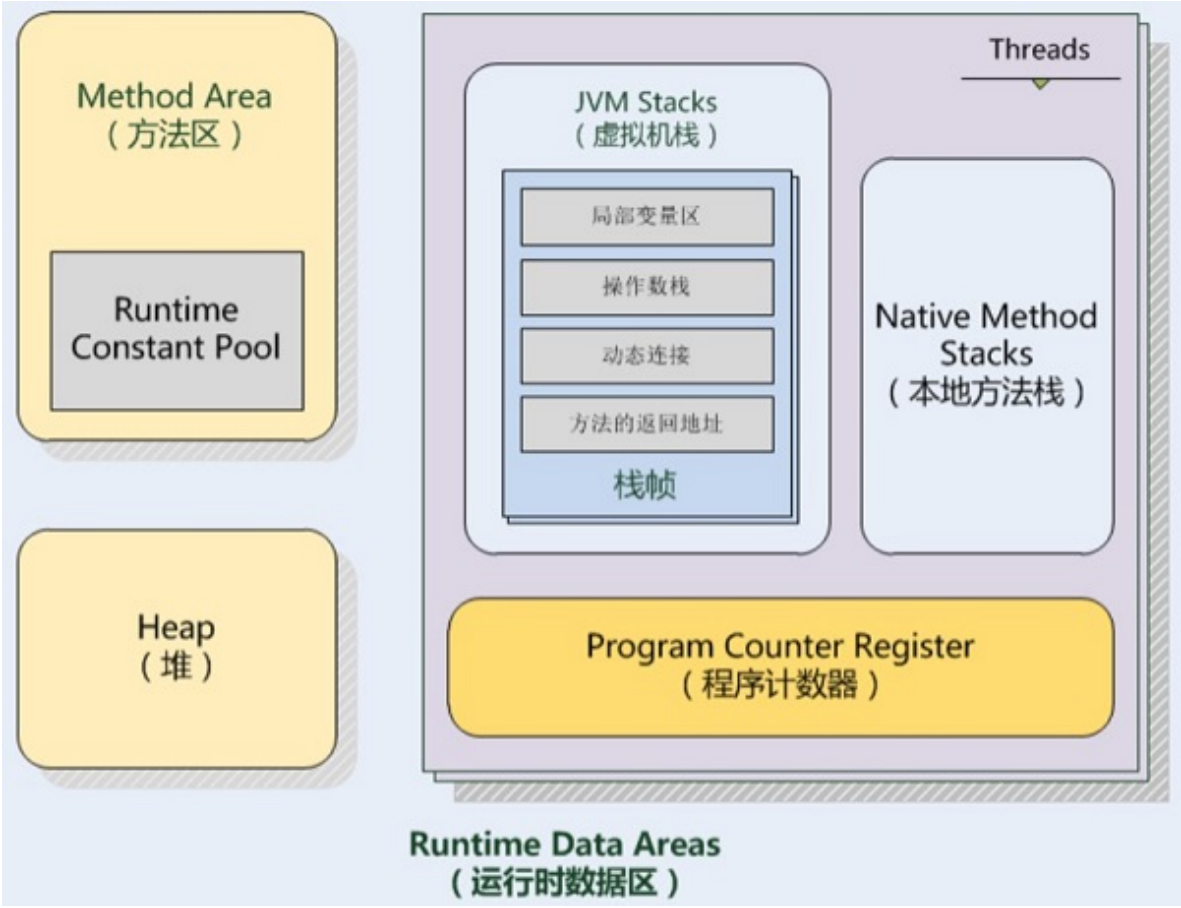
Java内存区域

我们平时所说的内存就是图中的运行时数据区（Runtime Data Area），其中与字符串的创建有关的是方法区（Method Area）、堆区（Heap Area）和栈区（Stack Area）。

方法区：存储类信息、常量、静态变量。全局共享。

堆区：存放对象和数组。全局共享。

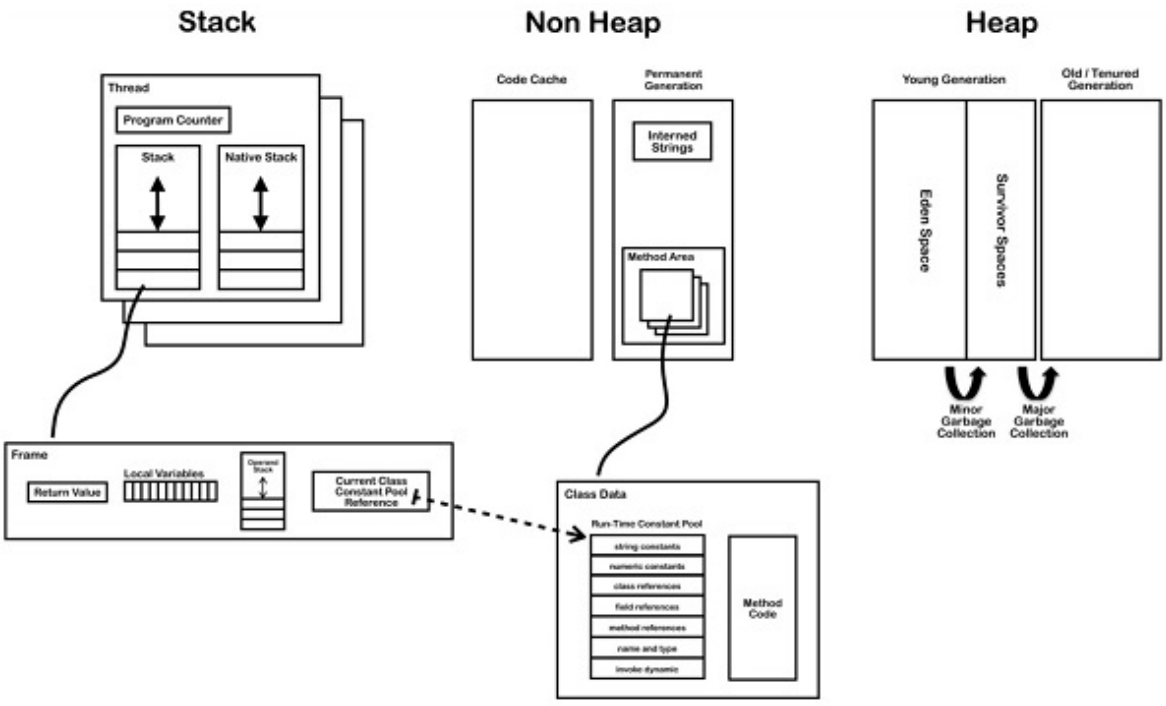
栈区：基本数据类型、对象的引用都存放在这。线程私有。



每当一个方法被执行时就会在栈区中创建一个栈帧（Stack Frame），基本数据类型和对象引用就存在栈帧中局部变量表（Local Variables）。

当一个类被加载之后，类信息就存储在非堆的方法区中。在方法区中，有一块叫做运行时常量池（Runtime Constant Pool）,它是每个class文件中的“常量池”被加载器加载之后就映射存放在这，后面会说到这一点。

和String最相关的是字符串池（String Pool），其位置在方法区上面的驻留字符串（Interned Strings）的位置，之前一直把它和运行时常量池搞混，其实是两个完全不同的存储区域，字符串常量池是全局共享的。字符串调用String.intern()方法后，其引用就存放在String Pool中。



两种创建方式在内存中的区别

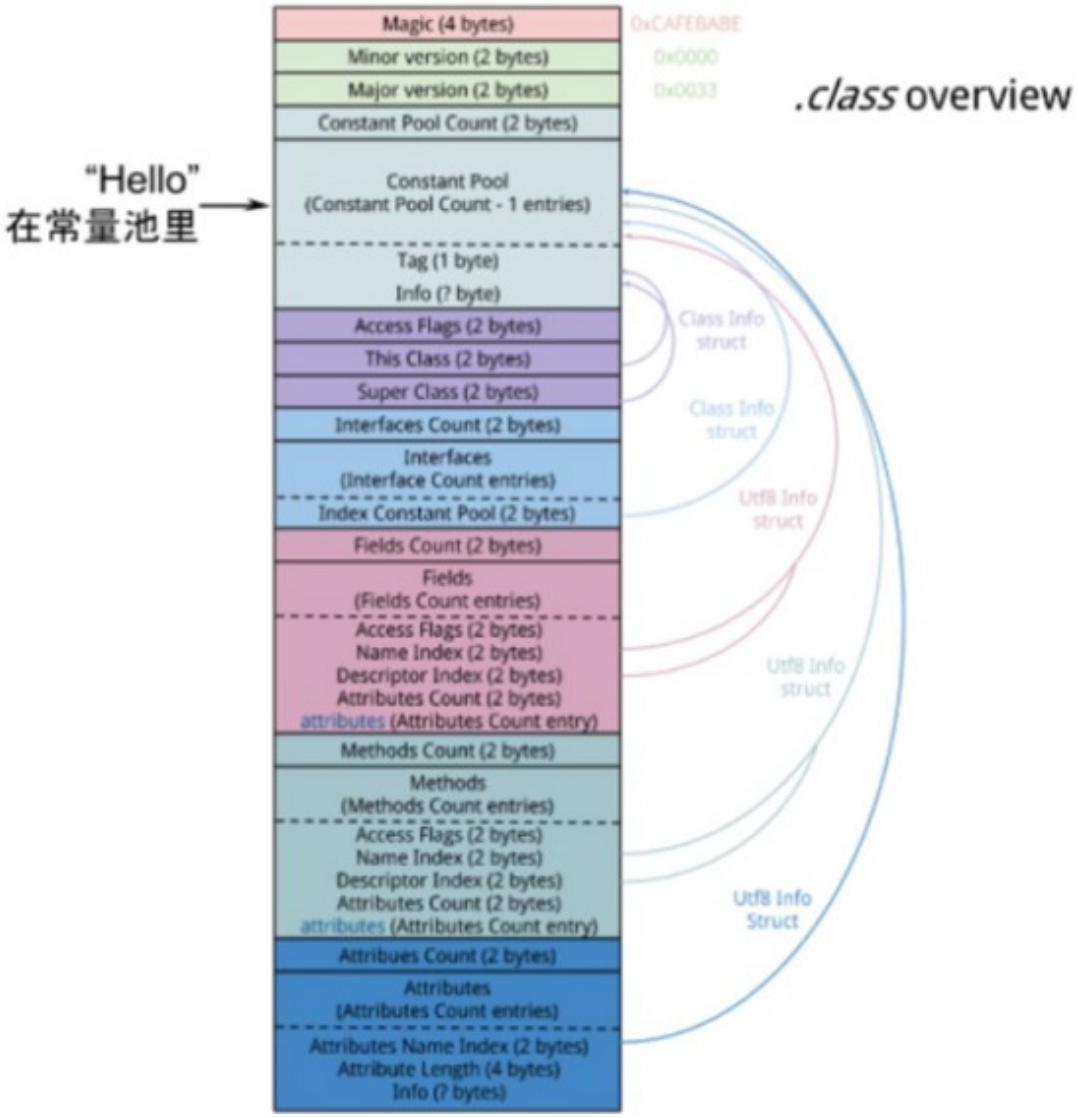
了解了这些概念，下面来说说究竟两种字符串创建方式有何区别。

下面的Test类，在main方法里以“字面量”赋值的方式给字符串str赋值为“Hello”。

```
public class Test {
    public static void main(String[] args) {

        String str = "Hello";

    }
}
```

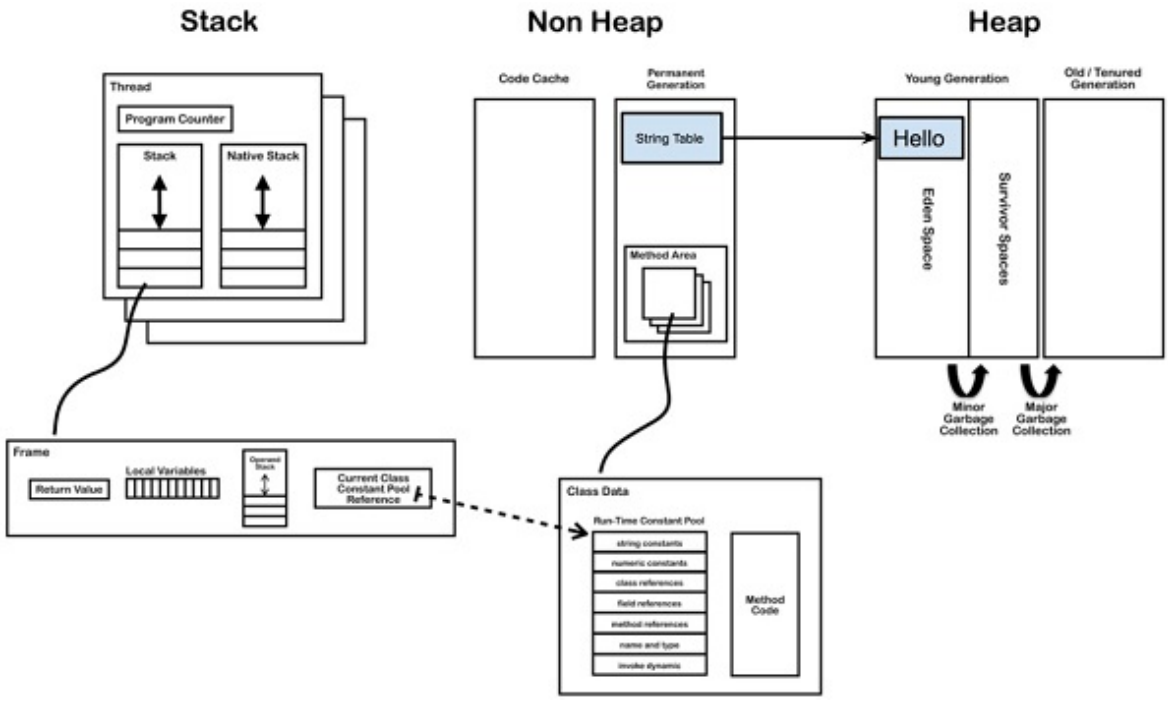
Test.java文件编译后得到.class文件，里面包含了类的信息，其中有一块叫做常量池（Constant Pool）的区域，.class常量池和内存中的常量池并不是一个东西。

.class文件常量池主要存储的就包括字面量，字面量包括类中定义的常量，由于String是不可变的，所以字符串“Hello”就存放在这。



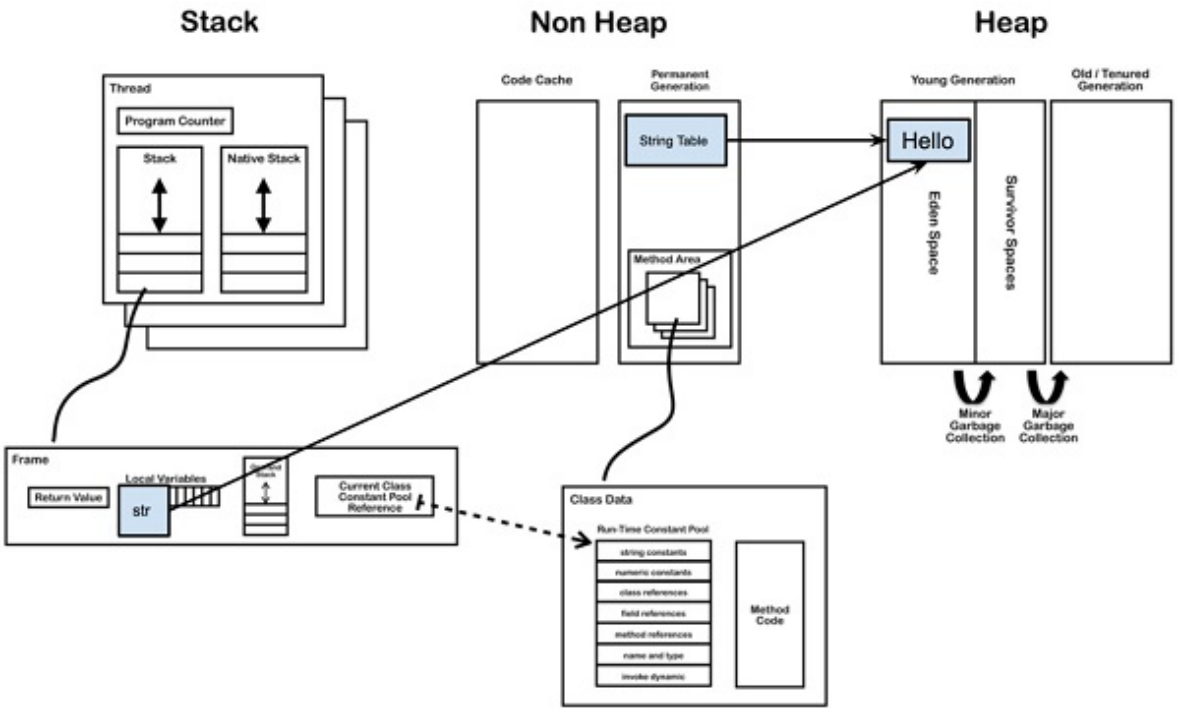
当程序用到Test类时，Test.class被解析到内存中的方法区。.class文件中的常量池信息会被加载到运行时常量池，但String不是。

例子中“Hello”会在堆区中创建一个对象，同时会在字符串池（String Pool）存放一个它的引用，如下图所示。



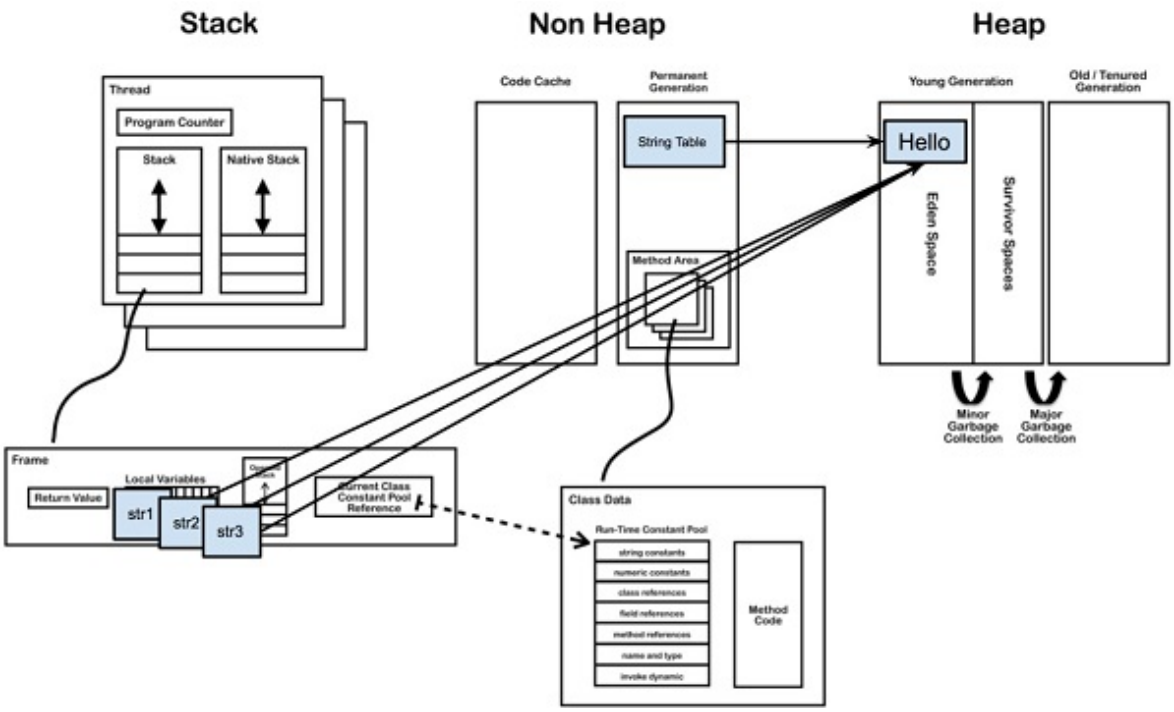
此时只是Test类刚刚被加载，主函数中的str并没有被创建，而“Hello”对象已经创建在于堆中。

当主线程开始创建str变量的，虚拟机会去字符串池中找是否有equals(“Hello”)的String，如果相等就把在字符串池中“Hello”的引用复制给str。如果找不到相等的字符串，就会在堆中新建一个对象，同时把引用驻留在字符串池，再把引用赋给str。

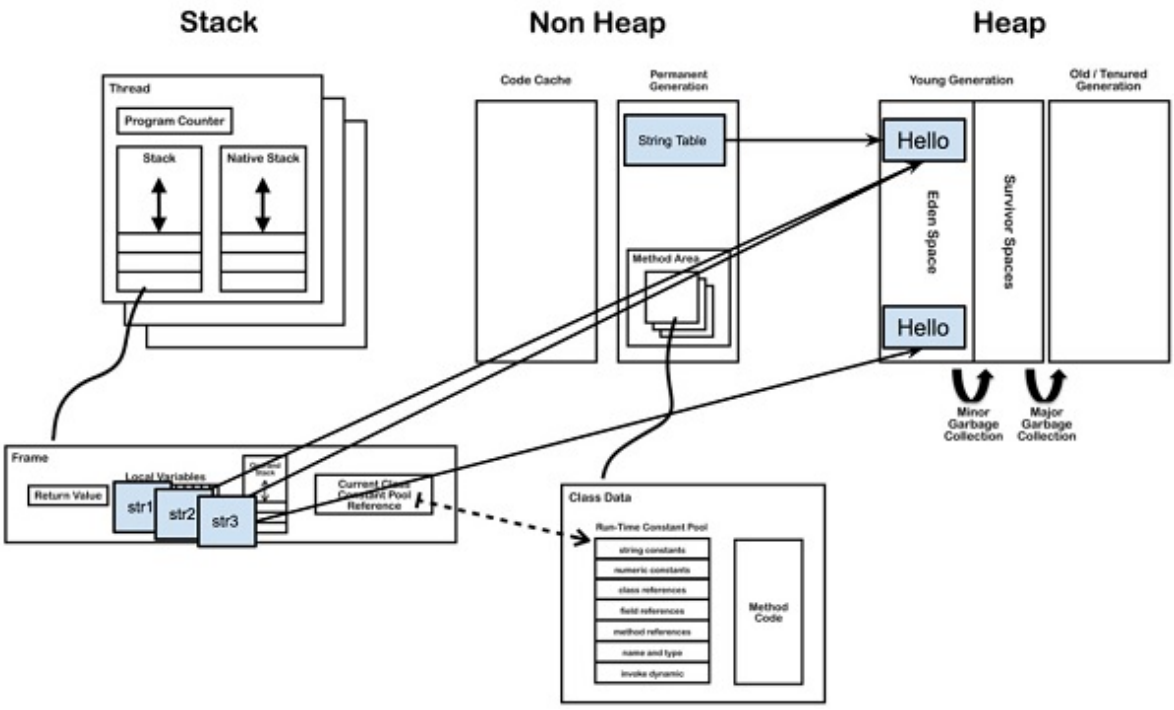


//当用字面量赋值的方法创建字符串时，无论创建多少次，只要字符串的值相同，它们所指向的都是堆中的同一个对象。

```
public class Test {
    public static void main(String[] args) {

        String str1 = "Hello";
        String str2 = "Hello";
        String str3 = "Hello";

    }
}
```




```
    //当利用new关键字去创建字符串时，前面加载的过程是一样的，只是在运行时无论字符串池中有没有与当前值相等的对象引用，都会在堆中新开辟一块内存，创建一个对象。

public class Test {
    public static void main(String[] args) {

        String str1 = "Hello";
        String str2 = “Hello”;
        String str3 = new String("Hello");

    }
}
```



代码阅读题：

```
String s1 = "Hello";
```

```
String s2 = "Hello";
String s3 = "Hel" + "lo";
String s4 = "Hel" + new String("lo");
String s5 = new String("Hello");
String s6 = s5.intern();
String s7 = "H";
String s8 = "ello";
String s9 = s7 + s8;

System.out.println(s1 == s2);  // true
System.out.println(s1 == s3);  // true
System.out.println(s1 == s4);  // false
System.out.println(s1 == s9);  // false
System.out.println(s4 == s5);  // false
System.out.println(s1 == s6);  // true
```

有了上面的基础，这个问题就迎刃而解了。

s1在创建对象的同时，在字符串池中也创建了其对象的引用。

由于s2也是利用字面量创建，所以会先去字符串池中寻找是否有相等的字符串，显然s1已经帮他创建好了，它可以直接使用其引用。那么s1和s2所指向的都是同一个地址，所以s1==s2。

s3是一个字符串拼接操作，参与拼接的部分都是字面量，编译器会进行优化，在编译时s3就变成“Hello”了，所以s1==s3。

s4虽然也是拼接，但“lo”是通过new关键字创建的，在编译期无法知道它的地址，所以不能像s3一样优化。所以必须要等到运行时才能确定，必然新对象的地址和前面的不同。

同理，s9由两个变量拼接，编译期也不知道他们的具体位置，不会做出优化。

s5是new出来的，在堆中的地址肯定和s4不同。

s6利用intern()方法得到了s5在字符串池的引用，并不是s5本身的地址。由于它们在字符串池的引用都指向同一个“Hello”对象，自然s1==s6。

总结一下：

字面量创建字符串会先在字符串池中找，看是否有相等的对象，没有的话就在堆中创建，把地址驻留在字符串池；有的话则直接用池中的引用，避免重复创建对象。

new关键字创建时，前面的操作和字面量创建一样，只不过最后在运行时会创建一个新对象，变量所引用的都是这个新对象的地址。

由于不同版本的JDK内存会有些变化，JDK1.6字符串常量池在永久代，1.7移到了堆中，1.8用元空间代替了永久代。但是基本对上面的结论没有影响，思想是一样的

经典面试题

String str=new String("abc");创建了几个String对象？

1、思路：我们可以把上面这行代码分成String str、=、”abc”和new String()四部分来看待。

String str只是定义了一个名为str的String类型的变量，因此它并没有创建对象；

=是对变量str进行初始化，将某个对象的引用赋值给它，显然也没有创建对象；

new String(“abc”)为什么又能被看成”abc”和new String()呢？我们来看一下被我们调用了的String的构造器：

```
        Copied!

public String(String original) {
    this.value = original.value;
    this.hash = original.hash;
}
```

我们知道我们常用的创建一个类的实例（对象）的方法有以下两种：

使用new创建对象。

调用Class类的newInstance方法，利用反射机制创建对象

我们正是使用 new 调用了 String 类的上面的构造器方法创建了一个对象，并将它的引用赋值给了str变量。同时我们注意到，被调用的构造器方法接受的参数也是一个String对象，这个对象正是”abc”。由此我们又要引入另外一种创建String对象的方式的讨论——引号内包含文本。

这里我们需要引入对字符串池相关知识：

```
String str="abc";//创建一个对象

String a="abc";
String b="abc";//创建一个对象

String c="ab"+"cd";//创建三个对象
```

在JAVA虚拟机（JVM）中存在着一个字符串池，其中保存着很多String对象，并且可以被共享使用，因此它提高了效率。由于String 类是final的，它的值一经创建就不可改变，因此我们不用担心String对象共享而带来程序的混乱。字符串池由String类维护，我们可以调用 intern()方法来访问字符串池。

我们再回头看看String a=”abc”;，这行代码被执行的时候，JAVA虚拟机首先在字符串池中查找是否已经存在了值为”abc”的这么一个对象，它的判断依据是String 类equals(Object obj)方法的返回值。如果有，则不再创建新的对象，直接返回已存在对象的引用；如果没有，则先创建这个对象，然后把它加入到字符串池中，再将它的引用返 回。因此，我们不难理解前面三个例子中头两个例子为什么是这个答案了。

对于第三个例子，“ab”和”cd”分别创建了一个对象，它们经过“+”连接后又创建了一个对象”abcd”，因此一共三个，并且它们都被保存在字符串池里了。

现在问题又来了，是不是所有经过“+”连接后得到的字符串都会被添加到字符串池中呢？我们都知道“==”可以用来比较两个变量，它有以下两种情况：

如果比较的是两个基本类型（char，byte，short，int，long，float，double，boolean），则是判断它们的值是否相等。

如果表较的是两个对象变量，则是判断它们的引用是否指向同一个对象。

回归正题：

答案应该是：两个或一个。

1、如果 abc 字符串之前没有用过，这毫无疑问创建了两个对象，一个是new String 创建的一个新的对象，一个是常量“abc”对象的内容创建出的一个新的String对象；

2、如果 abc 字符串之前有用过，那么是创建一个对象。

扩展阅读

理解Java字符串常量池与intern()方法:
https://www.cnblogs.com/justcooooode/p/7603381.html

字符串常量池、class常量池和运行时常量池:
https://blog.csdn.net/qq_26222859/article/details/73135660
http://tangxman.github.io/2015/07/27/the-difference-of-java-string-pool/

为什么Java中的String类是不可变的?
https://www.cnblogs.com/justcooooode/p/7514863.html

【译】Java中的字符串字面量:
https://www.cnblogs.com/justcooooode/p/7670256.html

面试题之String str = new String("abc"); 创建了几个对象:
https://blog.csdn.net/limingchuan123456789/article/details/14150327
https://blog.csdn.net/u011033906/article/details/53858409

6、StringBuffer 字符串缓冲区

StringBuffer类概述

所在包: java.lang

类的定义:

```
public final class StringBuffer
    extends AbstractStringBuilder
    implements java.io.Serializable, CharSequence
```

线程安全的可变字符序列，JDK1.0中声明，可以对字符串内容进行增删，此时不会产生新的对象。虽然在任意时间点上它都包含某种特定的字符序列，但通过某些方法调用可以改变该序列的长度和内容。

可将字符串缓冲区安全地用于多个线程。可以在必要时对这些方法进行同步，因此任意特定实例上的所有操作就好像是以串行顺序发生的，该顺序与所涉及的每个线程进行的方法调用顺序一致。

StringBuffer 上的主要操作是 append 和 insert 方法，可重载这些方法，以接受任意类型的数据。每个方法都能有效地将给定的数据转换成字符串，然后将该字符串的字符追加或插入到字符串缓冲区中。append 方法始终将这些字符添加到缓冲区的末端；而 insert 方法则在指定的点添加字符。

通常，如果 sb 引用 StringBuffer 的一个实例，则 sb.append(x) 和 sb.insert(sb.length(), x) 具有相同的效果。

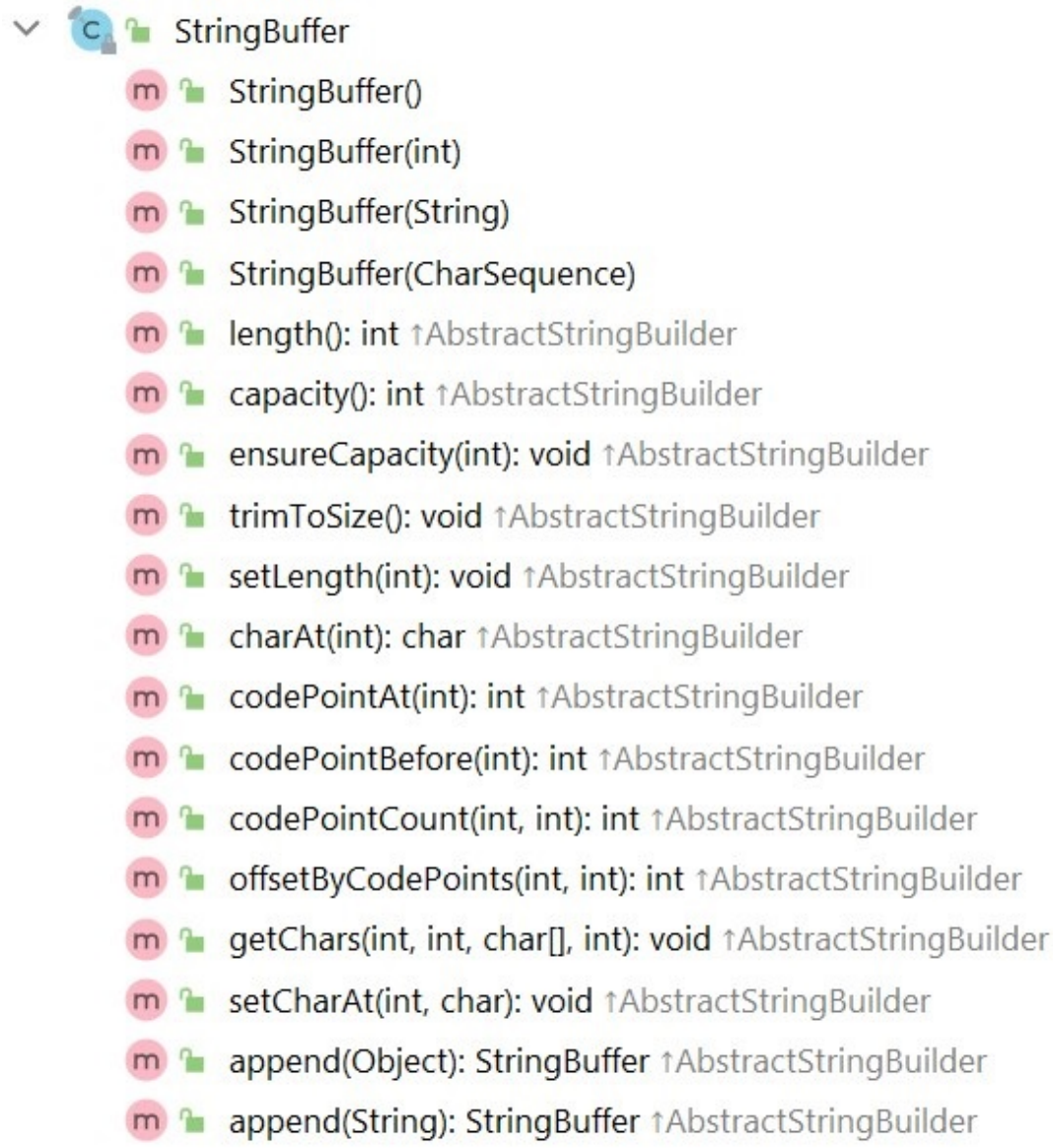
当发生与源序列有关的操作（如源序列中的追加或插入操作）时，该类只在执行此操作的字符串缓冲区上而不是在源上实现同步。

每个字符串缓冲区都有一定的容量。只要字符串缓冲区所包含的字符序列的长度没有超出此容量，就无需分配新的内部缓冲区数组。如果内部缓冲区溢出，则此容量自动增大。从 JDK 5 开始，为该类补充了一个单个线程使用的等价类，即 StringBuilder。与该类相比，通常应该优先使用 StringBuilder 类，因为它支持所有相同的操作，但由于它不执行同步，所以速度更快。

特点:

1、可以对字符串内容进行修改
2、是一个容器
3、是可变长度的
4、缓冲区中可以存储任意类型的数据
5、最终需要变成字符串

StringBuffer类结构图



方法摘要

在StringBuffer类中存在很多和String类一样的方法，这些方法在功能上和String类中的功能是完全一样的。

但是有一个最显著的区别在于，对于StringBuffer对象的每次修改都会改变对象自身，这点是和String类最大的区别。

作为参数传递时，方法内部可以改变值。

```
  abstract class AbstractStringBuilder implements Appendable, CharSequence {
    /**
     * The value is used for character storage
     * value没有final声明,value可以不断扩容。
     */
    char[] value;

    /**
     * The count is the number of characters used.
     * count记录有效字符的个数。
     */
    int count;
}
```

构造方法

```
    StringBuffer()  构造一个其中不带字符的字符串缓冲区，其初始容量为 16 个字符。

    StringBuffer(CharSequence seq)  public java.lang.StringBuilder(CharSequence seq) 构造一个字符串缓冲区，它包含与指定的 CharSequence 相同的字符。

    StringBuffer(int capacity)  构造一个不带字符，但具有指定初始容量的字符串缓冲区。

    StringBuffer(String str)    构造一个字符串缓冲区，并将其内容初始化为指定的字符串内容。
```

成员方法

添加

```
    StringBuffer append(各种数据类型的参数)  将各种数据类型的参数的字符串表示形式追加到序列。
        参数类型：
            boolean b：将 boolean 参数的字符串表示形式追加到序列。
            char c：将 char 参数的字符串表示形式追加到此序列。
            char[] str：将 char 数组参数的字符串表示形式追加到此序列。
            char[] str, int offset, int len：将 char 数组参数的子数组的字符串表示形式追加到此序列。
            CharSequence s：将指定的 CharSequence 追加到该序列。
            CharSequence s, int start, int end：将指定 CharSequence 的子序列追加到此序列。
            double d：将 double 参数的字符串表示形式追加到此序列。
            float f：将 float 参数的字符串表示形式追加到此序列。
            int i：将 int 参数的字符串表示形式追加到此序列。
            long lng：将 long 参数的字符串表示形式追加到此序列。
            Object obj：追加 Object 参数的字符串表示形式。
            String str：将指定的字符串追加到此字符序列。
            StringBuffer sb：将指定的 StringBuffer 追加到此序列中。

    StringBuffer appendCodePoint(int codePoint) 将 codePoint 参数的字符串表示形式追加到此序列。

    StringBuffer insert(int offset, 各种数据类型的参数)  将各种数据类型的参数的字符串表示形式插入此序列中。
        参数类型和append一致；
```

删除

```
    StringBuffer delete(int start, int end) 移除此序列的子字符串中的字符。

    StringBuffer deleteCharAt(int index)    移除此序列指定位置的 char。
```

修改

```
    void ensureCapacity(int minimumCapacity)    确保容量至少等于指定的最小值。

    StringBuffer replace(int start, int end, String str)    使用给定 String 中的字符替换此序列的子字符串中的字符。

    void setCharAt(int index, char ch)  将给定索引处的字符设置为 ch。

    void setLength(int newLength)   设置字符序列的长度。

    void trimToSize()   尝试减少用于字符序列的存储空间。
```

查找

```
    int indexOf(String str) 返回第一次出现的指定子字符串在该字符串中的索引。

    int indexOf(String str, int fromIndex)  从指定的索引处开始，返回第一次出现的指定子字符串在该字符串中的索引。

    int lastIndexOf(String str) 返回最右边出现的指定子字符串在此字符串中的索引。

    int lastIndexOf(String str, int fromIndex)  返回最后一次出现的指定子字符串在此字符串中的索引。
```

获取子串

```
    CharSequence subSequence(int start, int end)    返回一个新的字符序列，该字符序列是此序列的子序列。

    String substring(int start) 返回一个新的 String，它包含此字符序列当前所包含的字符子序列。

    String substring(int start, int end)    返回一个新的 String，它包含此序列当前所包含的字符子序列。
```

反转

```
    StringBuffer reverse()  将此字符序列用其反转形式取代。
```

方法原理解析

当append和insert时，如果原来value数组长度不够，可扩容。
如上这些方法支持方法链操作。

方法链的原理：

```
@Override
public synchronized StringBuffer append(Object obj) {
```

```
    toStringCache = null;
    super.append(String.valueOf(obj));
    return this;
}

@Override
public synchronized StringBuffer insert(int offset, String str) {
    toStringCache = null;
    super.insert(offset, str);
    return this;
}
```

7、StringBuilder 字符串缓冲区

StringBuilder类概述

所在包:java.lang

类的定义:

```
public final class StringBuilder
    extends AbstractStringBuilder
    implements java.io.Serializable, CharSequence
```

一个可变的字符序列。此类提供一个与 StringBuffer 兼容的 API，但不保证同步。该类被设计用作 StringBuffer 的一个简易替换，用在字符串缓冲区被单个线程使用的时候（这种情况很普遍）。如果可能，建议优先采用该类，因为在大多数实现中，它比 StringBuffer 要快。

在 StringBuilder 上的主要操作是 append 和 insert 方法，可重载这些方法，以接受任意类型的数据。每个方法都能有效地将给定的数据转换成字符串，然后将该字符串的字符追加或插入到字符串生成器中。append 方法始终将这些字符添加到生成器的末端；而 insert 方法则在指定的点添加字符。

将 StringBuilder 的实例用于多个线程是不安全的。如果需要这样的同步，则建议使用 StringBuffer。

方法摘要

Stringbuilder和StringBuffer的方法一致;只是一个是线程安全的；一个是线程不安全的;

不再一一列举方法;

深入理解String、StringBuffer和StringBuilder类的区别

Java提供了String、StringBuffer和StringBuilder类来封装字符串，并提供了一系列操作字符串对象的方法。

它们的相同点是都用来封装字符串；都实现了CharSequence接口。它们之间的区别如下：

可变与不可变

String类是一个不可变类，即创建String对象后，该对象中的字符串是不可改变的，直到这个对象被销毁。

StringBuffer与StringBuilder都继承自AbstractStringBuilder类，在AbstractStringBuilder中也是使用字符数组保存字符串，是可变类。

由于String是不可变类，适合在需要被共享的场合中使用，当一个字符串经常被修改时，最好使用StringBuffer实现。如果用String保存一个经常被修改的字符串，该字符串每次修改时都会创建新的无用的对象，这些无用的对象会被垃圾回收器回收，会影响程序的性能，不建议这么做。

初始化方式

当创建String对象时，可以利用构造方法String str = new String("Java")的方式来对其进行初始化，也可以直接用赋值的方式String s = "Java"来初始化。

而StringBuffer与StringBuilder只能使用构造方法方式初始化。

字符串修改方式

String字符串修改方法是首先创建一个StringBuilder，其次调用StringBuilder的append方法，最后调用StringBuilder的toString()方法把结果返回，示例代码如下：

```
String str = "hello";
str += "java";
```

以上代码等价于下面的代码：

```
StringBuilder sb = new StringBuilder(str);
sb.append("java");
str = sb.toString();
```

上述String字符串的修改过程要比StringBuffer多一些额外操作，会增加一些临时的对象，从而导致程序的执行效率降低。StringBuffer和StringBuilder在修改字符串方面比String的性能要高。

是否实现了equals和hashCode方法

String实现了equals()方法和hashCode()方法，new String("java").equals(new String("java"))的结果为true；
而StringBuffer没有实现equals()方法和hashCode()方法，因此，new StringBuffer("java").equals(new StringBuffer("java"))的结果为false，将StringBuffer对象存储进Java集合类中会出现问题。

是否线程安全

StringBuffer与StringBuilder都提供了一系列插入、追加、改变字符串里的字符序列的方法，它们的用法基本相同，只是StringBuilder是线程不安全的，StringBuffer是线程安全的。如果只是在单线程中使用字符串缓冲区，则StringBuilder的效率会高些，但是当多线程访问时，最好使用StringBuffer。

```
public class CompareTimeTest {
    public static void main(String[] args) {
        //初始设置
```

```
        long startTime = 0L;
        long endTime = 0L;
        String text = "";
        StringBuffer buffer = new StringBuffer("");
        StringBuilder builder = new StringBuilder("");

        //开始对比
        startTime = System.currentTimeMillis();
        for (int i = 0; i < 20000; i++) {
            buffer.append(String.valueOf(i));
        }
        endTime = System.currentTimeMillis();
        System.out.println("StringBuffer的执行时间：" + (endTime - startTime));

        startTime = System.currentTimeMillis();
        for (int i = 0; i < 20000; i++) {
            builder.append(String.valueOf(i));
        }
        endTime = System.currentTimeMillis();
        System.out.println("StringBuilder的执行时间：" + (endTime - startTime));

        startTime = System.currentTimeMillis();
        for (int i = 0; i < 20000; i++) {
            text = text + i;
        }
        endTime = System.currentTimeMillis();
        System.out.println("String的执行时间：" + (endTime - startTime));
    }
}
```

综上所述

综上，在执行效率方面，StringBuilder最高，StringBuffer次之，String最低，对于这种情况，一般而言，如果要操作的数量比较小，应优先使用String类；如果是在单线程下操作大量数据，应优先使用StringBuilder类；如果是在多线程下操作大量数据，应优先使用StringBuffer类。

6、递归

递归方法：一个方法体内调用它自身。

方法递归包含了一种隐式的循环，它会重复执行某段代码，但这种重复执行无须循环控制。

递归一定要向已知方向递归，否则这种递归就变成了无穷递归，类似于死循环。

示例代码：

```
//计算1-100之间所有自然数的和
public int sum(int num){
    if(num == 1){
        return 1;
    }else{
        return num + sum(num - 1);
    }
}
```

递归方法练习题：

```
    1、请用Java写出递归求阶乘(n!)的算法

        Copied!

// 计算1-n之间所有自然数的乘积:n!
public int getProduct(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * getProduct(n - 1);
    }

}

        Copied!

    2、已知有一个数列：f(0) = 1,f(1) = 4,f(n+2)=2*f(n+1) + f(n),其中n是大于0的整数，求f(10)的值。

        Copied!

public int f(int n){
    if(n == 0){
        return 1;
    }else if(n == 1){
        return 4;
    }else{
        return 2*f(n - 1) + f(n - 2);
    }
}
```