2020红岩网校移动开发部

Android第四节课 - Java基础的复习和补充

> 终于来到了第四节课！过了这节课就跳出冰冷的黑框框，打开Android应用程序的大门了~~
>
> 你到现在为止学的东西都是工具~ 为了下节课学习Android提供基础、能更好更快地进行“安卓手机APP”的开发~
>
> 感谢昔日勤奋好学的你，坚持到了现在。你辛苦敲下的代码中的每一个字母，熬夜中掉的每一根头发，都将会在学习安卓前路中支撑你，让你变得更优秀。
>
> 兴趣是最好的老师，保持对一个事物的热爱，你就会不断地去探索，在不知不觉中就：
>
> 
>
>
> 请记住：“放弃，是人生中最大的遗憾和失败”。
>
> 不多bb了，接下来让我们来进入本次的课程吧~~

0x01 Java 基础部分复习

先复习下一些Java的名词：

> **关键字：**int、float、main、if、for、while、public、private、instanceof
>
> **标识符：**int tongShen，这里tongShen就是个标识符
>
> **运算符：**像这种+、-、*、/...
>
> **变量：**由用户声明的变量，比如：int a;

到这里是不是很简单？那再加点料😁

> **类：**类是一个模板，它描述一类对象的行为和状态。例如：

```
class Dog{
    int age;
    void bark(){
        System.out.println("汪汪汪");
    }
}
```

> 请问这样生成了一个小狗吗？并没有。
>
> **对象：**对象是类的一个实例（对象是类实例化的结果）。例如，一条中华田园犬是一个对象，它的状态有：颜色、名字、品种；行为有：摇尾巴、叫、吃等。

Dog dog = new Dog(); 这样才生成了一个小狗

然后才能执行这样的语句：

dog.age = 3;

dog.bark()

是不是很像：int number = 10；这种写法呀?

> **实例化**：将类去具体化生成一个变量（或称对象）（类只是个模板）。
>
> **你可以简单地理解"对象是一个复杂的变量"**

那既然是变量，为什么要new后才能使用呢？？？

这是java的机制，java为了高效运行，将变量分为了两大类：

> 1、基本类型（int、float、double...）
>
> 2、引用类型（对象）

内存空间也分了两部分：

> 1、栈空间（哈哈先别管为啥叫这个名字，学了大计基就懂了）
>
> 2、堆空间

栈空间里放的是基本数据类型，堆里面放对象。

实际上我们 Dog dog = null;

这样只是在栈中创建一个像int一样的小变量，存一个整数（地址）

然后，new Dog()告诉java去堆空间中开辟一段内存空间，将这个内存空间的首地址给dog变量。

> **方法**：方法就是行为，一个类可以有很多方法。逻辑运算、数据修改以及所有动作都是在方法中完成的。在C语言中叫函数，在Java中叫方法，实际上是同一个东西。
> **实例变量**：每个对象都有独特的实例变量，对象的状态由这些实例变量的值决定。
>
> **静态**：你可能听过许多"静态方法"、"静态变量"这些叫法吧？为啥"静态方法不能调用非静态方法"，今天我就给你讲清楚。

静态的变量就是在运行一次程序时，自始至终都只存在一个的变量。

如：

```
class Dog{
    static int instanceCounter = 0;
}
```

关键字 static 就是表示声明的这个变量是静态的。

这个时候，instanceCounter就是一个静态变量，不管 new Dog() 实例化Dog类多少次，instanceCounter这个变量始终只有一个。通过[类名].[静态变量]这种格式访问。静态变量又叫类变量。

与静态变量对应的是实例变量。[类名].[实例变量]是不能访问这个变量的。只有[对象].[实例变量]这样子访问。所以现在可以理解了吗：实例变量可以有多个（取决你new了多少次），而类变量始终只有一个。

静态方法、实例方法以此类推鸭~

有了上面的基础知识后，现在再反过来看我们以前常写的main方法：

```
class Main{
    public static void main(String[]){
        // Writing your code ...
    }
}
```

java规定，静态main方法是程序的起点（一切java程序都是由main方法跑起来的）

我们刚刚说了，方法是依附于类而存在的（没有类就没有方法，跟c语言不同）。所以方法要写在类中，上面的写法就懂了吧?

那静态函数为啥不能调用实例函数呢?

```
class Main{
    int a;
    void printHelloWorld(){
        System.out.println("Hello World!");
    }
    public static void main(String[] args){
        // 你可能想：
        printHelloWorld();
        // 或者想：
        Main.printHelloWorld(); // 错误：静态方法不能调用实例方法
        a = 666; // 错误：静态方法不能访问实例变量

        // 正确的写法：
        Main main = new Main(); // 实例化
        main.printHelloWorld(); // 调用实例方法
        main.a = 666;
    }
}
```

静态方法在java刚运行就存在了，而这时没有实例化，所以不存在实例函数，同样也不存在实例变量。

0x02 Java 继承复习

> 类继承存在的意义实际上就是：
>
> 让我们在原有类的基础上，快速构造出新的类（类型），这样省去了很多代码量，并且提供了一些方便的关键字去管理类。
>
> 接口可以初步地简单理解为："扩展函数"。一个类如果实现了这个接口，那么，它就获得了接口中所有的函数，就像扩展了一样。

让我们回忆一下类继承的语法：

```
interface BarkAbility{
    void bark();
}

class Animal{
    int age;
    Animal(int age){
        this.age = age;
    }
}
```

```
class Dog implements BarkAbility{
    @Override
    public void bark(){
        System.out.println("汪汪汪！");
    }
    void run(){
        System.out.println("跑步中");
    }
}

class Fish{
    void swim(){
        System.out.println("游泳中");
    }

}

class Main{
    public static void main(String[] args){
        Dog dog = new Dog();
        Fish fish = new Fish();

        fish.swim();
        dog.run();
        dog.bark();

        if(fish instanceof BarkAbility){
            ((BarkAbility) fish).bark();
        }

        if(dog instanceof BarkAbility){
            ((BarkAbility) dog).bark();
        }

    }
}
```

0x03 Java 多线程（Thread）

截止目前为止！我们写的程序都是“运行在主线程”的“单线程程序”。多线程的意思是 将一个代码分交给几个人执行。充分发挥cpu的多核性能。

我们通过例子来理解什么是线程：

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

        // Writing your code1 ...

        // 运行到这里之前，还是一个人在处理事情
        // 到这里，新建了一个 “Thread 线程” 对象，并且start了，相当于叫另一个人说：“来！帮我处理下这件事（指code2）”
        new Thread(){
            @Override
            public void run() {

                // Writing your code2 ...
```

```
            }
        }.start();

        // Writing your code3 ...

    }
}
```

并行：多个CPU同时执行多个任务，比如：多个人同时做不同的事
并发：一个CPU（采用时间片）同时执行多个任务。比如：前一毫秒还在做进程1的事情，后一毫秒就在做进程2的事情了。

进程：是程序的一次执行过程，或是正在运行的一个程序，是一个动态的过程。对于java来说，就是main函数的执行过程。
线程：进程可进一步细化为线程，是一个程序内部的一条执行路径（比如说java的某几行代码，同时执行，这里就要用到线程）。

线程的详细用法

```
Thread.currentThread().getName(); // 获得当前的线程的名字
```

上面是什么写法呢?

类名+静态方法，这样获得了一个实例，再调用实例的getName方法。

currentThread这个静态方法比较特殊，java会自动获得“正在执行的这一行代码所在的进程实例”。

实例方法：start()

> 1.启动当前线程　2.调用线程实例中的run方法

实例方法：run()

> 一个线程实例中，需要重写Runnable接口中的run方法。
>
> 然后调用实例.run方法相当于直接在本线程中执行run方法中的内容。

实例方法：getName() / setName(String name)

> 获取/设置当前进程的名字

静态方法：yield()

> 主动释放当前所在的线程的执行权
>
> 给cpu调度，比较容易让给别的进程，但是也存在自己继续执行的情况

实例方法：join()

> 在线程中插入执行另一个线程，该线程被阻塞，直到插入执行的线程完全执行完毕以后，该线程才继续执行下去。
>
> 注：与run()方法不同的是，这样执行实际上是开了一个新的线程。而直接调用run()方法会执行在本进程。

实例方法：stop()

> 强行停止进程

静态方法：sleep(long ms)

> 线程阻塞一段时间（毫秒）

实例方法：isAlive()

> 判断当前线程是否存活

```
class Main extends Thread{

    Main(String name){
        super(name);
    }

    @Override
    public void run() {
        for (int i = 0; i < 30; i++) {
            System.out.println(getName() + i);
        }
    }

    public static void main(String[] args){
        Thread thread1 = new Main("煜姐==");
        Thread thread2 = new Main("潼姐-----");
        thread1.start();
        thread2.start();
    }
}
```

> 更深入的线程理解，请参考：
>
> https://blog.csdn.net/pange1991/article/details/53860651

巩固一下接口回调，其实很简单：

```
interface Callback{
    void doAfter(String result);
}

class Main{

    public static void main(String[] args){

        System.out.println("自己做事情1");

        Callback callback = new Callback() {
            @Override
            public void doAfter(String result) {
                // 网络请求后，通常要将数据显示出来，这里就做显示的事情。
                System.out.println("result is:" + result);
            }
        };

        new Thread(){
            @Override
            public void run() {
                System.out.println("叫人代做事情3");
                try{ // try - catch 语法见下文“异常”
```

```
                    sleep(2000);
                }catch (InterruptedException e){
                    e.printStackTrace();
                }
                callback.doAfter("别人做完了事情3");
            }
        }.start();

        System.out.println("自己做事情2");

    }
}
```

其实这里的callback变量，实际上是final类型的（即不可变类型的）。

上节课煜姐提到内部类如果要访问外部类中的变量，则只能使用不能修改（这里涉及到闭包的知识，省略500字，如感兴趣请点击下方链接）。在jdk1.8后就不需要加final了，实际上只是加入了个“语法糖”，省略了这个关键字，实际上这个变量还是不能修改的。

语法糖是啥呀？就是一种简便的写法，让程序员感到甜的语法

> 欲详细了解 内部类访问外部变量为啥要加final，以及啥是闭包，请移步：
>
> https://www.zhihu.com/question/21395848

0x04 Java 泛型

泛型在java中有很重要的地位，在面向对象编程及各种设计模式中有非常广泛的应用。

什么是泛型？为什么要使用泛型？

> 泛型，即“参数化类型”。一提到参数，最熟悉的就是定义方法时有形参，然后调用此方法时传递实参。那么参数化类型怎么理解呢？
>
> 顾名思义，就是将类型由原来的具体的类型参数化，类似于方法中的变量参数，此时类型也定义成参数形式（可以称之为类型形参），
>
> 然后在使用/调用时传入具体的类型（类型实参）。
>
> 泛型的本质是为了参数化类型（在不创建新的类型的情况下，通过泛型指定的不同类型来控制形参具体限制的类型）。也就是说在泛型使用过程中，
>
> 操作的数据类型被指定为一个参数，这种参数类型可以用在类、接口和方法中，分别被称为泛型类、泛型接口、泛型方法。

在学习泛型之前，我们先聊聊int和Integer的区别。

数据类型	字节大小	封装类
byte	8位	Byte
short	16位	Short
int	32位	**Interger**
long	64位	Long
float	32位	Float
double	64位	Double
boolean	1位	Boolean

- 1、自动装箱
  自动装箱其实就是将基本数据类型转换为引用数据类型（对象）
- 2、自动拆箱
  自动拆箱其实就是将引用数据类型转化为基本数据类型

> Java int和Integer的区别详解：
>
> https://blog.csdn.net/tongsiw/article/details/89851213

现在我们继续泛型的学习：

```
ArrayList<String> strings = new ArrayList<>();
strings.add("666");
// 不知道这个语法大家使用过吗？
```

这种尖括号就是典型的“泛型”语法。

一个数组，如果我们事先写好一个存String类型的数组（专门存字符串的），那么它就不能存整数、浮点数了。

```
class StringList{
    String[] list = new String[200];
    int size = 0;
    void append(String s){
        list[size] = s;
    }
    String get(int index){
        return list[index];
    }
    void remove(int index){
        // ...
    }
    void insert(int index){
        // ...
    }
}
```

如果我们要用这个类去盛装整数类型的变量，我们就要这样修改：

```
class IntList{
    int[] list = new int[200];
    int size = 0;
    void append(int s){
        list[size] = s;
    }
    int get(int index){
        return list[index];
    }
    void remove(int index){
        // ...
    }
    void insert(int index){
        // ...
    }
}
```

这样子这个类写了两次，如果需要修改，则两个类需要同时修改，有什么办法让用户指定“所存放的数据类型呢”?

这就是泛型存在的意义，“参数化类型”，将类型也参数化，提高通用性，其此之谓乎!

```
class MyList<T>{
    Object[] list = new Object[200];
    int size = 0;
    void append(T s){
        list[size] = s;
    }
    T get(int index){
        return (T)list[index];
    }
    void remove(int index){
        // ...
    }
    void insert(int index){
        // ...
    }
}
```

泛型类的基本写法:

```
class 类名称 <泛型标识>{
    泛型标识 var1;
    int var2
    泛型标识 get(泛型标识 var){
        // ...
    }
}
```

泛型标识可以是任意标识，我们常用T、E、K、V作为泛型标识。

注意：泛型必须是包装类型或者类名，不能是基本数据类型。

泛型标识就像一个变量一样，可以让用户自由选择使用哪个类型。

泛型类与泛型方法

```
class DataSave <T> {
    private T data;
    public T getData(){ // 这不是泛型方法
        return data;
    }
    public void setData(T data){
        this.data = data;
    }
    // 这才叫泛型方法，通俗来说，你给我什么类型的参数，我的K就是什么类型的，我就返回什么类型的值。
    public <K> K get(K hh){
        return hh;
    }
}

class Main{
    public static void main(String[] args) {
        DataSave<String> dataSave = new DataSave<>();
        dataSave.setData("数据");
        System.out.println("保存的数据为：" + dataSave.getData());
        System.out.println("泛型方法输出为：" + dataSave.get(6f));
    }
}
```

泛型接口

```
interface MyInterface<T> {
    public T get();
}

// 那么如何使用这个泛型接口呢？
// 方式1：(给接口传入一个确定的类型)
class A implements MyInterface<String>{
    @Override
    public String get() {
        return null;
    }
}

// 方式2：（将这个类也变成泛型类，让用户指定接口的泛型是哪个类型）
class A <T> implements MyInterface<T>{
    @Override
    public T get() {
        return null;
    }
}
```

上限通配 下限通配

我们先建立水果抽象类：

```
/**
 * 水果类
 */
public abstract class Fruit {
    public abstract void eat();
}
```

再用两个具体的水果去继承它：

```
/**
 * 苹果类
 */
public class Apple extends Fruit {
    @Override
    public void eat() {
        System.out.println("我是苹果，我是酸酸甜甜的");
    }
}

/**
 * 香蕉类
 */
public class Banana extends Fruit {
    @Override
    public void eat() {
        System.out.println("我是香蕉，我是软软的");
    }
}
```

这时我们还要个吃瓜群众：

```
/**
 * 吃瓜群众
 */
public class People <T extends Fruit> {
    public void eatFruit(T t){
        t.eat();
    }
}
```

这里的的意思是：接受一个泛型，这个泛型的类型 必须是Fruit的子类（也包括Fruit本身）。（上限通配）

以此类推，的意思是：接受一个泛型，这个泛型的类型 必须是Fruit的父类（也包括Fruit本身）。（下限通配，较少用）

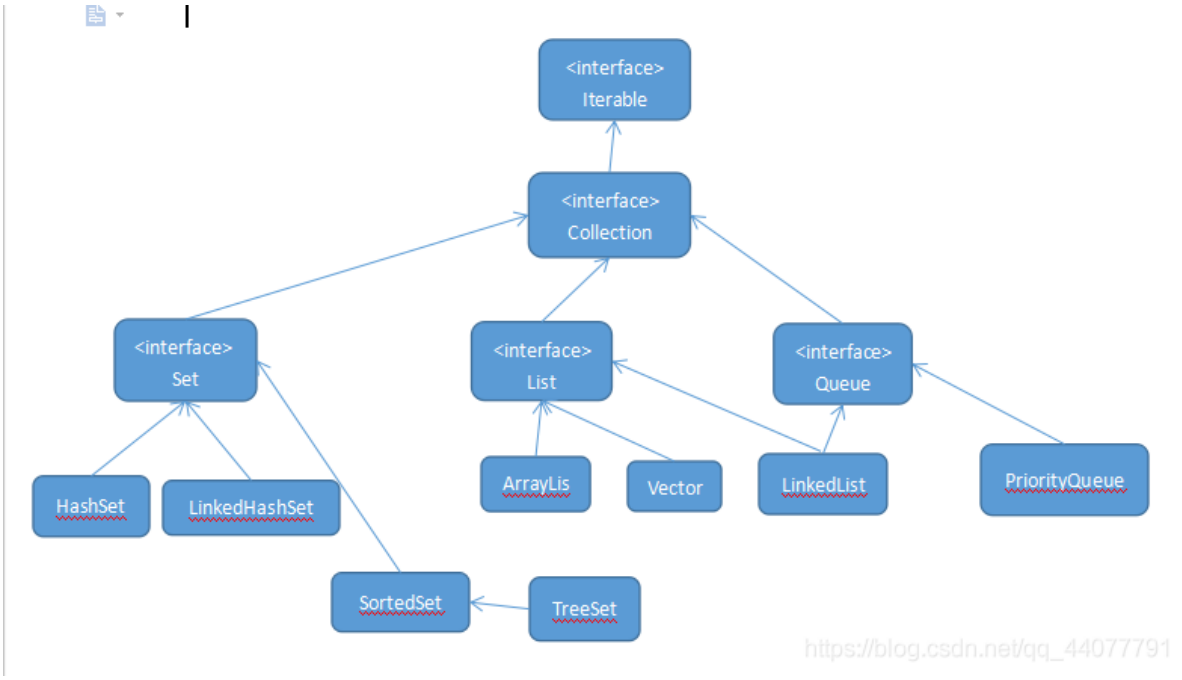
通配符与泛型擦除

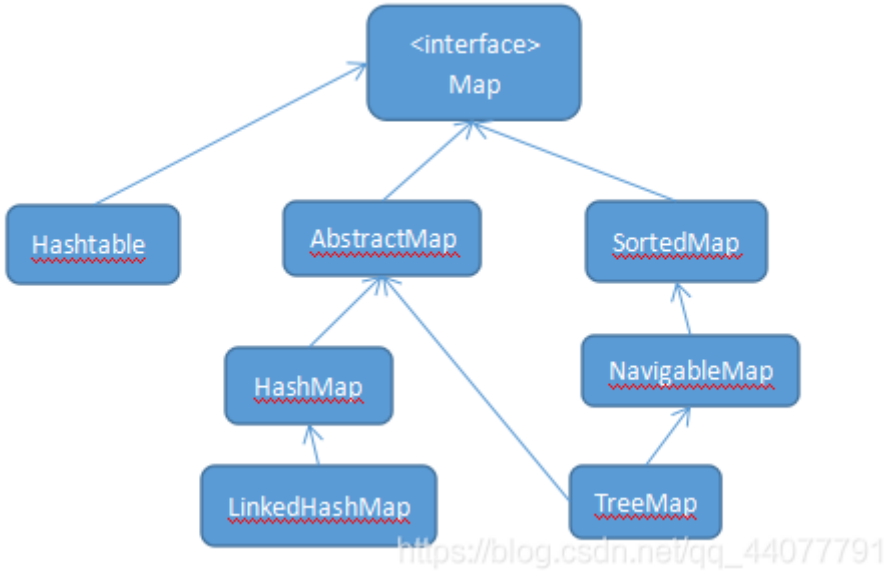
> 属于进阶内容，感兴趣可以自行了解~

0x05 Java 集合

> 附一篇超详细的Java集合的教程：（有许多继承关系，适合大佬看）

> https://blog.csdn.net/feiyanaffection/article/details/81394745






```
// 我们现在会：
int[] numbers = new int[100];
```

如果我们想要向数组中插入一个数，那么其后的每个数字都需要往后移动，这可要花费不少时间，效率会很低。

如果我们在数组后面追加一堆数字，则需要判断size是否大于100，如果大于100了，还需要重新分配内存：

```
numbers2 = new int[200];
```

还需要把原来的数据全部用循环复制过来。

种种迹象表明，在实际应用中，数组不是个很好的选择。

而java之所以火爆，也依赖于java有许多高效的“集合类”，使得数据处理变得如此简单~

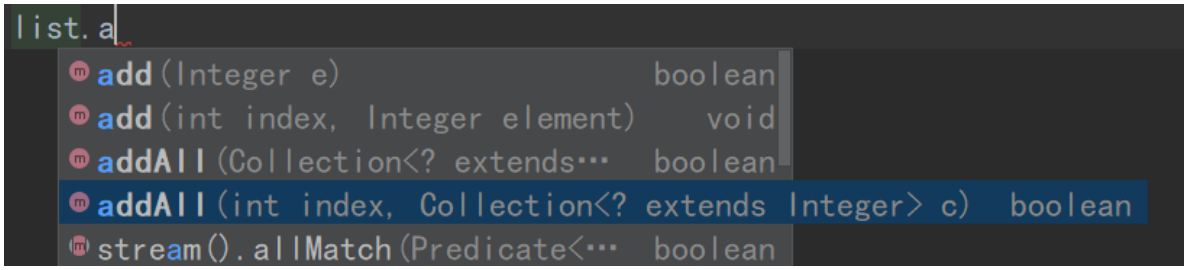
我们如何建立一个动态数组呢：

```
ArrayList<String> list = new ArrayList<>();
```

这时，这个数组的大小就是动态可变的了。（还没忘泛型吧哈哈哈）

我们可以：

```
list.add("hello");
list.add(int index, "hahaha"); // 在指定位置插入元素
list.get(int index);
list.remove(int index); // 删除指定位置的元素，后面的元素会自动补位
list.remove(Object obj); // 遍历数组，找到与obj匹配的元素，删除它
list.size(); // 返回数组的大小
list.contains(Object obj); // 返回是否包含与obj匹配的元素
list.addAll(int index, Collection<? extends Integer> c); // 将集合c中的每个元素插入到list的index处
```



用addAll来复习一下 类继承 和 我们刚刚学的泛型~

我们查阅上面的继承关系表，可以看到：ArrayList是实现Collection接口的，而其他许许多多的类也是实现了Collection接口的。addAll这个方法就是Collection接口中的方法，所以ArrayList可以调用addAll这个方法。而这个方法的参数接收一个Collection，也就是说，所有实现了Collection接口的类的对象，都可以作为参数，给addAll方法使用。

我们看看实现了Collection的类有哪些：

Set（集合，此集合非彼集合，此集合是数学意义上的集合，即：一个集合里不能有相同的元素，所以我们一般拿集合去重）：

HashSet、LinkedHashSet、SortedSet、TreeSet（这些类的底层实现不一样，所以在效率方面各有所长，使用起来是一样的。）

List（动态数组，随意增删查改）：

ArrayList（线程不安全）、Vector（线程安全，但是效率很低）、LinkedList（虽然继承List接口，可以对任意索引的元素进行访问，但实际上内存中是双向链表实现的，所以查找元素很慢）

Queue（队列，在数据结构的意义上是：先进先出，可以理解为数组只允许在末尾增加元素，在头部删除元素）：

LinkedList（普通队列，用双向循环链表实现）、PriorityQueue（优先队列，插入元素时会自动按自然顺序将元素排列，非常高效）

Map（键值对表，详见下文）

其中注意到LinkedList既实现了Queue接口，也实现了List接口，即“既允许在头部删除、在末尾添加，又允许像普通数组一样增删查改”。

所以通过这个案例，我们可以更进一步的理解面向对象存在的意义。

比如说：ArrayList和Vector都是实现了List接口的“实现类”对吧？

```
// List接口
public interface List<E> extends Collection<E> {
}
// 我们可以联想一下刚刚讲的泛型和这个写法，加深对泛型的理解。
```

```
// 因为他们俩都是接口的实现类，于是可以这么写：
List<String> list1 = new ArrayList<>();
List<String> list2 = new Vector<>();
// 可以看到，一个“父类型”，竟然可以装两个不同类型的对象，可见面向对象的灵活性
```

化为一般的形式就是：

```
接口名 对象 = new 该接口的实现类();
// 这样子，生成的“对象”，就只能用“接口里的方法”了。
```

而我们也可以继续联想一下，抽象类是不是也可以这样。

详细的接口中的方法请自行查阅文档，或访问上文给出的链接

注意到所有的这些类都实现了Iterable接口（迭代器）。

啥是迭代器呢?

看看Iterable接口中的方法：

> Object next()：返回下一个元素，每次调用next之前，应该先检查是否有下一个元素hasNext()
> boolean hasNext()：判断容器中是否还有可供访问的元素
> void remove()：删除迭代器刚越过的元素

这些方法，实际上提供给我们一个可以遍历所有元素的办法。这些方法的实现都是在“实现类 的内部”写好的，我们无需关心这个方法是如何实现的。

```
ArrayList<String> list = new ArrayList<>();
// 我们可以这样遍历数组中每个元素：
for (int i = 0; i < list.size(); i++){
    System.out.println(list.get(i));
}
// 通过本次学习，我们还可以这样遍历：
Iterator iterator = list.iterator();
while (iterator.hasNext()){
    System.out.println(iterator.next().getName());
}
```

还记得涛涛学姐第一节课讲的for-each语法吗?

```
ArrayList<String> list = new ArrayList<>();
for (String item : list){
    System.out.println(item);
}
// 抽象出来的for-each语法是这样的：
// for (你要遍历的集合的元素类型 局部变量名称 : 你要遍历的集合){
//     ...
// }
```

这个写法的底层原理就是调用的Iterable接口的方法，是java的语法糖hhh

也就是说，所有继承了可迭代（Iterable）接口的集合都能使用for-each语法~

Map是啥

键值对表。什么是键值对？key-value形式，建立一个key对value的映射，在数学上我们知道x→y的映射为f(x)，实际上在java的map中，x就是key、y就是value

这么说可能不大懂，接下来用例子演示：

```
Map<Integer, String> map = new HashMap<>();

map.put(2, "我是二号"); // 相当于放了一个映射关系：当x为2，则f(x)为“我是二号”
map.put(3, "我是三号");
map.put(3, "我不对劲"); // 重复的键，值会覆盖。

System.out.println(map.get(3));
```

实际上可以简单理解为：

我们平时用的数组的下标都必须是整数对吧？

map其实也是一个数组，只不过它的下标可以是对象（上面的对象实际上就是Integer）。数组是建立数与数之间的映射，map是建立一种对象与对象的映射。

底层实现原理是啥呢？根据key对象的地址，通过哈希算法（计算出一个地址），然后List[地址] = value对象地址（这里就是传统意义上的数组了）。

想了解的可以下课找我battle哈，还是挺好玩的hhhh。。

而集合实际上用的时候就查查文档就好了，我们实际开发中使用的多的集合类有：

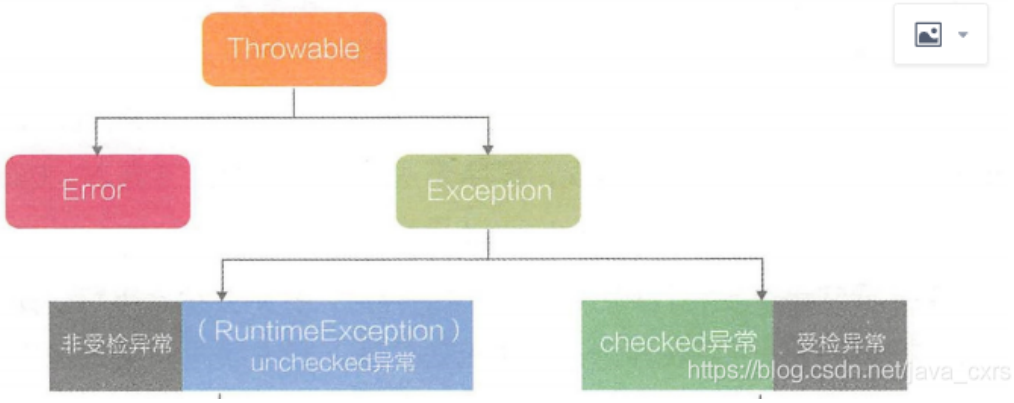
HashMap、ArrayList、HashSet

功能很简单，详细其实翻阅“对应接口”的注释，就可以了解个八九。

0x06 Java 异常

> 当程序出现各种各样的错误时，称这些错误为异常。

异常的分类：



> Error：是程序中无法处理的错误，表示运行应用程序中出现了严重的错误。此类错误一般表示代码运行时JVM出现问题。通常有Virtual MachineError（虚拟机运行错误）、NoClassDefFoundError（类定义错误）等。比如说当jvm耗完可用内存时，将出现OutOfMemoryError。此类错误发生时，JVM将终止线程。非代码性错误。因此，当此类错误发生时，不应该去处理此类错误。
>
> Exception：：程序本身可以捕获并且可以处理的异常。

我们重点讨论Exception。

Exception也可以分为两类：受检异常和非受检异常

> 运行时异常(不受检异常)：RuntimeException类极其子类表示JVM在运行期间可能出现的错误。编译器不会检查此类异常，并且不要求处理异常，比如用空值对象的引用（NullPointerException）、数组下标越界（ArrayIndexOutBoundException）。此类异常属于不可查异常，一般是由程序逻辑错误引起的，在程序中可以选择捕获处理，也可以不处理。
>
> 非运行时异常(受检异常)：Exception中除RuntimeException极其子类之外的异常。编译器会检查此类异常，如果程序中出现此类异常，比如说IOException，必须对该异常进行处理，要么使用try-catch捕获，要么使用throws语句抛出，否则编译不通过。

简而言之，能被编译器提示出来的，叫受检异常，反之叫非受检异常。

如果是受检异常，则只有两种处理方法：

1、使用try - catch语句，将异常“捕获”

2、在函数头加上throws [异常名]，像调用者抛出异常，则调用者调用这个函数时，也会有异常，必须重复1或2部。

不然是无法通过编译器检测的，也就无法编译运行。

```
/** try - catch的基本语法 **/
try {
    // 这里乱写代码都可以hhh，执行时遇到错误立刻跳转到catch语句（不会中断java程序的运行）。
} catch (Exception e) {
    // 这句话加上表示将遇到的错误输出到控制台上。如果不加这句话，表示jvm默默承受（不输出到控制台）
    e.printStackTrace();
} finally {
    // 可以不加finally这句，finally属于进阶用法，请自行了解~
}
```

所有的异常都继承自Exception，所以上述代码直接捕获Exception，这时，如果try语句中抛出的是Exception的子类，则都能捕获到。

```
try {
    // 这里遇到哪个错误，就跳到哪个catch块中执行。就像选择结构一样。
}catch (InterruptedException e){
    e.printStackTrace();
}catch (IOException e){
    e.printStackTrace();
}
```

如何自己写一个异常?

```
class MyException extends Exception{ }
```

```
class Main{

    static int num = 0;

    static int getNumber() throws MyException{
        if(num == 0){
            throw new MyException();
        }else{
            return num;
        }
    }

    public static void main(String[] args) {
        try {
            System.out.println(getNumber());
        }catch (MyException e){
            e.printStackTrace();
        }
    }
}
```

> 讲的很透彻的一篇：加finally和不加finally的区别：
>
> https://blog.csdn.net/weixin_42965795/article/details/103056065

0x07 Java 反射

听起来好高大上，hhh。

我们知道一个类中private的方法是在外部无法访问的对吧，private的变量也无法访问。

现在我告诉你，其实“java反射机制”，就可以让你像开外挂一样访问到某个类的私有属性和方法~

> Java反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意方法和属性；这种动态获取信息以及动态调用对象方法的功能称为Java语言的反射机制。
>
> 非常详细的教程：https://blog.csdn.net/qq_36226453/article/details/82790375

之前不是说：java变量有两种类型：基本类型和引用类型吗？

引用类型实际上说的就是对象（实例）。

而这些对象其实也可以分为两大类：实例对象和Class对象。

Class对象实际上是java运行时保存每个类的运行状态、基本信息的对象（当你的程序中执行到出现一个类的类名时，jvm虚拟机就会去读取这个类的文件，并且加载到内存中来），不同于我们的new，这些对象是自动创建的

而“运行状态、基本信息”包含什么呢？当然是所有的变量、所有的方法都保存在这个class对象里啦~

所以我们通过得到类对应的class对象，就能调用它的私有方法、修改它的私有变量了，是不是像开外挂一样。

而我们的实例对象，也是通过class对象创建的。在运行期间，一个类只会有一个class对象产生，不会产生第二个。

那么如何去获得这个对象呢？（获取的是引用）

三种方式：

```
/** 方式1 **/
Student stu = new Student(); // 这时产生了一个Student对象和一个Class对象
Class stuClass = stu.getClass(); // 获取Class对象

/** 方式2 **/
Class stuClass = Student.class;

/** 方式3 **/
Class stuClass = Class.forName("reflect.Student"); // 包名.类名 的形式
```

Class类的主要方法（以下全部为实例方法）

- String **getName**()：获得类的完整名字（包名.类名）。
- Field[] **getFields**()：获得类的public类型的属性。
- Field **getField**(String name)：获得类的指定的public类型的属性。
- Field[] **getDeclaredFields**()：获得类的所有属性。
- Field **getDeclaredField**(String name)：获得类的指定的属性。
- Method[] **getMethods**()：获得类的所有public类型的方法。
- Method **getMethod**(String name, Class[] parameterTypes)：获得类的特定方法，name参数指定方法的名字，parameterTypes参数指定方法的参数类型。
- Method[] **getDeclaredMethods**()：获得类的所有方法。
- Method **getDeclaredMethod**(String name, Class[] parameterTypes)：获得类的特定方法。
- Constructor<?>[] **getConstrutors**()：获得类的所有public类型的构造方法。
- Constructor< T> **getConstrutor**(Class[] parameterTypes)：获得类的特定的public的构造方法，parameterTypes参数指定构造方法的参数类型。
- Constructor<?>[] **getDeclaredConstrutors**()：获得类的所有的构造方法。
- Constructor< T> **getDeclaredConstrutor**(Class[] parameterTypes)：获得类的特定的构造方法，parameterTypes参数指定构造方法的参数类型。
- T **newInstance**()：通过类的**不带参数的构造方法**创建这个类的一个对象。

注：jdk9之后class.newInstance()过时，可以用class.getDeclaredConstructor().newInstance()代替。有参构造可以调用 `class.getDeclaredConstructor(String.class, int.class).newInstance("张三", 123456);` 两个参数为姓名和学号。

```
class Student{
    String name;
    int id;
    private static int instanceCounter = 0;
    private void study(String subject){
        System.out.println(name + id + " 我在偷学：" + subject);
    }
    private Student(String name, int id){
        this.name = name;
        this.id = id;
        instanceCounter++;
    }
    public static int getInstanceCounter(){
        return instanceCounter;
    }
}

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

```
        Class<Student> stuClass = Student.class;

        try {
            /** 通过获取私有构造方法创建实例 **/
            Constructor<Student> studentConstructor =
                stuClass.getDeclaredConstructor(String.class, int.class);
            studentConstructor.setAccessible(true);
            // 调用非public的东西，必须要先设置为可使用，下同
            Student student = studentConstructor.newInstance("张三", 123456);

            /** 获取私有方法并运行 **/
            Method study = stuClass.getDeclaredMethod("study", String.class);
            study.setAccessible(true);
            study.invoke(student, "数学");

            /** 获取静态有返回值方法并运行 **/
            Method getCount = stuClass.getDeclaredMethod("getInstanceCounter");
            System.out.println("Student对象数量: " + getCount.invoke(null));

            /** 获取静态私有变量，并修改值 **/
            Field count = stuClass.getDeclaredField("instanceCounter");
            count.setAccessible(true);
            count.set(student, 9);

            /** 验证是否修改成功 **/
            System.out.println("Student对象数量: " + getCount.invoke(null));

        }catch (Exception e){
            e.printStackTrace();
        }
    }
}
```

注：若用getDeclaredMethods()，则不仅会获取到当前类的方法，还会获取到父类中的方法。