java8

:: method references

There are four kinds of method references:

- Reference to a **static method** `ClassName::staticMethodName`
- Reference to an **instance method** of a particular object `Object::instanceMethodName`
- Reference to an instance method of an arbitrary object of a particular type `ContainingType::methodName` –
- Reference to a constructor `ClassName::new`

```
List<String> list = Arrays.asList("node", "java", "python", "ruby");
list.forEach(System.out::println);          // method references


List<String> list = Arrays.asList("node", "java", "python", "ruby");
list.forEach(str -> System.out.println(str)); // lambda
```

java对象

object类的方法

- toString 。 该方法返回一个代表该对象的字符串。该方法的默认实现返回的字符串在绝大多数情况下是没有信息量的，因此通常都需要在子类中重写该方法。
- equals。该方法检验两个对象是否相等。该方法的默认实现使用 `==` 运算符检验两个对象是否相等，通常都需要在子类中重写该方法。
    - `==` 运算符，比较的是值是否相等，如果是对象，则比较的是对象所指向的内存地址是否相等。
    - 重写该方法，可以根据对象的值来判定是否相等。

      ```
      #Object中equals()的默认实现
      public boolean equals(Object obj) {
          return (this == obj);
      }
      ```

- hashCode 。该方法返回对象的散列码。
    - public native int hashCode() ， `native` 方法表示实现方法的编程语言不是 Java
    - 散列码是一个整数，用于在散列集合中存储并能快速查找对象。
    - 根据散列约定，如果两个对象相同，它们的散列码一定相同，因此如果在子类中重写了 equals 方法，必须在该子类中重写 hashCode 方法，以保证两个相等的对象对应的散列码是相同的。
    - 两个相等的对象一定具有相同的散列码，两个不同的对象也可能具有相同的散列码。实现 hashCode 方法时，应避免过多地出现两个不同的对象也可能具有相同的散列码的情况。

- finalize。该方法用于垃圾回收。如果一个对象不再能被访问，就变成了垃圾。`finalize` 方法会被该对象的垃圾回收程序调用。该方法的默认实现不做任何事，如果必要，子类应该重写该方法
- getClass。该方法返回对象的元对象。元对象是一个包含类信息的对象。包括类名、构造方法和方法等
- clone 。该方法用于复制一个对象，创建一个有单独内存空间的新对象
  - 不是所有的对象都可以复制，只有当一个类实现了 java.lang.Cloneable 接口时，这个类的对象才能被复制。
  - 该方法可能抛出 CloneNotSupportedException 异常。
  - new 对象和clone对象的区别。

    > - 程序执行到new操作符时， 首先去看new操作符后面的类型，因为知道了类型，才能知道要分配多大的内存空间。
    > - 分配完内存之后，再调用构造函数，填充对象的各个域，这一步叫做对象的初始化。
    > - 构造方法返回后，一个对象创建完毕，可以把他的引用（地址）发布到外部，在外部就可以使用这个引用操纵这个对象
    >
    > new操作符的本意是分配内存。调用clone方法时，分配的内存和源对象（即调用clone方法的对象）相同，然后再使用源对象中对应的各个域，填充新对象的域， 填充完成之后，clone方法返回，一个新的相同的对象被创建，同样可以把这个新对象的引用发布到外部。

```
Person p = new Person(23, "zhang");
Person p1 = p;   #引用的复制，p和p1指向同一个内存地址

Person p = new Person(23, "zhang");
Person p1 = (Person) p.clone();  #对象的复制，p和p1指向的内存地址也不同
```

浅复制和深复制

- 浅复制：被复制对象的所有变量都含有与原来的对象相同的值，而所有的对其他对象的引用仍然指向原来的对象。换言之，浅复制仅仅复制所考虑的对象，而不复制它所引用的对象。
- 深复制：被复制对象的所有变量都含有与原来的对象相同的值，除去那些引用其他对象的变量。那些引用其他对象的变量将**指向被复制过的新对象**，而不再是原有的那些被引用的对象。换言之，深复制把要复制的对象所引用的对象都复制了一遍。

对象的比较

Java中提供了两种方式来使得对象可以比较大小，实现`Comparator`接口或者`Comparable`接口

- Comparator接口

  实现Comparator接口需要重写其中的compare()方法。

  ```
  int compare(T o1,T o2)
  ```

  根据第一个参数小于、等于或大于第二个参数分别返回负整数、零或正整数，通常使用-1, 0, +1表示。

  ```
  
  ```

```
public class Student {
    //省去Student类的定义过程..
    public static void main(String[] args) {
        Student stu1 = new Student("sakuraamy",20);
        Student stu2 = new Student("sakurabob",21);
        Student stu3 = new Student("sakura",19);

        ArrayList<Student> stuList = new ArrayList<>();
        stuList.add(stu1);
        stuList.add(stu2);
        stuList.add(stu3);

    //没有必要去创建一个比较器类 采用内部类的方式实现Comparator接口
        Collections.sort(stuList, new Comparator<Student>() {
            @Override
            public int compare(Student o1, Student o2) {
                return (o1.age<o2.age ? -1 : (o1.age == o2.age ? 0 : 1));
                //return o1.name.compareTo(o2.name);
            }
        });
    //或者使用lambda表达式
    //Collections.sort(stuList, (o1,o2)->o1.age-o2.age);
        System.out.println(stuList);
        //输出 [name:sakura age:19, name:sakuraamy age:20, name:sakurabob
age:21]
    }
}
```

- 以`able`结尾的接口都表示拥有某种能力。若是某个自定义类实现了comparable接口，则表示**该类的实例对象拥有可以比较的能力**

  实现comparable接口需要覆盖其中的compareTo()方法。

```
int compareTo(T o)

public class Student implements Comparable<Student>{
    //省去Student的构造过程

  //重写comparaTo方法 以age作为标准比较大小
    @Override
    public int compareTo(Student o) {
        return return (this.age<o.age ? -1 : (this.age == o.age ? 0 :
1));;//本类接收本类对象，对象可以直接访问属性(取消了封装的形式)
    }

    public static void main(String[] args) {
        Student stu1 = new Student("sakura",20);
        Student stu2 = new Student("sakura",21);
        Student stu3 = new Student("sakura",19);
    //TreeSet会对插入的对象进行自动排序，所以要求知道对象之间的大小
        TreeSet<Student> stuSet = new TreeSet<>();
        stuSet.add(stu1);
        stuSet.add(stu2);
        stuSet.add(stu3);
    //使用foreach(), lambda表达式输出stuSet中的值 forEach()方法从JDK1.8才开始有
        stuSet.forEach(stu->System.out.println(stu));
    }
```

```
}
/*
output:
name:sakura age:19
name:sakura age:20
name:sakura age:21
*/
```

java异常

Java 语言中所有错误或异常的超类（父类）为Throwable

Throwable ，是对所有异常进行整合的一个普通类。它的作用就是能够**提取保存在堆栈中的错误信息**。

- 成员方法：
  - public String getMessage()：返回此 throwable 的详细消息字符串
  - public String toString()：获取异常类名和异常信息。
  - public void printStackTrace()：获取异常类名和异常信息，以及异常出现在程序中的位置。
  - public void printStackTrace(PrintStream s)：通常用该方法将异常内容保存在日志文件中，以便查阅。
- 异常的分类：
  - 异常(Exception)
  - 编译时异常: 凡是Exception或者是Exception的子类都成为编译时异常,这种是可以解决的,一般编译不通过
  - 运行时异常: 凡是RuntimeException或者是RuntimeException的子类都成为运行时异常,这种也是可以解决的,一般都是代码不够严谨或者逻辑错误
  - 错误 (Error)

> 为什么将throwable设计为一个普通类？
>
> 1. 因为Throwable里面存储的是错误描述的对象体现，是字符串，只不过把这些字符串表示为一个对象
> 2. 子类的解决异常处理方法里都是构造方法，一个是无参构造，另外一个就是带字符串参数（对象）的带参构造，既然有了构造方法，那就可以直接从父类中直接提取所需要的异常处理信息从而来构造一个方法，这样可以省略子类很多代码
> 3. 普通类继承的话是直接继承，不能够改变，如果改变的话就会影响到父类，根据这个特点也可以很清楚的说明Throwable为什么是普通类，因为它压根就不需要子类重写方法，只需要提取信息便可。

> jvm处理异常的方式？
>
> 1. 打印错误信息（异常的类名，异常的信息，异常的位置，错误的行号）
> 2. 将程序停止

> 我们处理异常的方式
>
> 1. 使用try catch来捕获异常并作处理，使得程序继续正常执行下去。如果未捕获到，还是会交给jvm的，所以建议把Exception作为catch的参数类型放在异常处理格式的最后
> 2. 在开发中,有的时候我们没有权限处理该异常,我们不知道该如何处理异常,或者不想处理异常,这种情况下我们可以将异常抛出,抛出给调用者处理

> 3. throw和throws的作用都是将异常抛出给调用者或者虚拟机来处理，但是两者有个根本区别就是throw的是**异常对象**，而throws的是**异常类**。throw在方法体内出现,throws在方法的声明上
> 4. 抛出异常的处理方法千万不能抛出给JVM处理[主方法]，调用者最好都处理，编译时异常必须出来，运行时异常建议处理。

位数组

位数组的应用是用更少的内存构建hashmap进行判断某个数值是否存在在海量的数据中。

- 位数组的核心思想是用一个char来表示8个整数，char转换成2进制后，正好是8位长，每一位表示一个整数

  > 10011010，分别对应0,1,2,3,4,5,6,7，其中0存在，3存在，4存在，6存在，其他不存在。
  >
  > 一个char数组表示8*lenth个数字了，内存大大减少