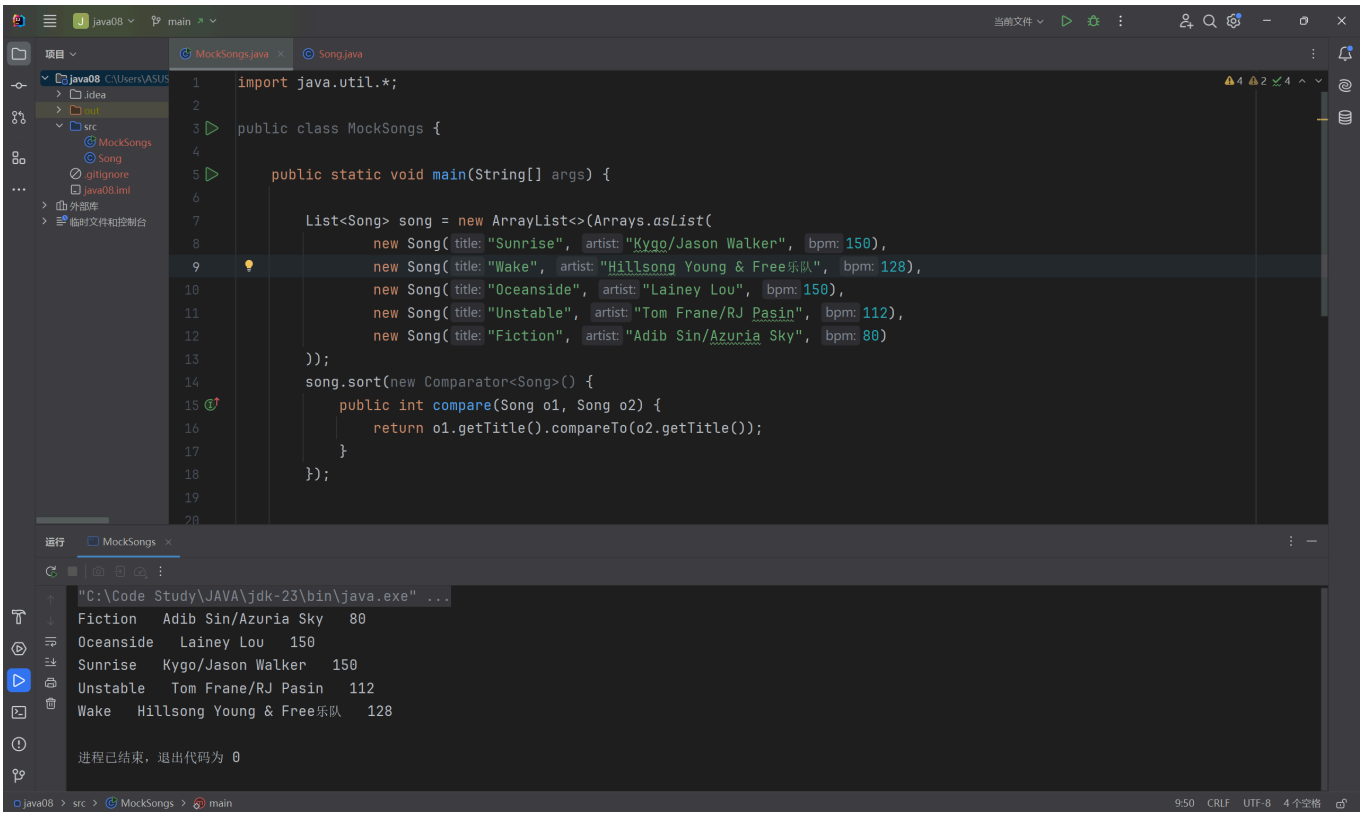
2024090905023-程宗健-JAVA08

```
import java.util.*;

public class MockSongs {

    public static void main(String[] args) {

        List<Song> song = new ArrayList<>(Arrays.asList(
                new Song("Sunrise", "Kygo/Jason Walker", 150),
                new Song("Wake", "Hillsong Young & Free乐队", 128),
                new Song("Oceanside", "Lainey Lou", 150),
                new Song("Unstable", "Tom Frane/RJ Pasin", 112),
                new Song("Fiction", "Adib Sin/Azuria Sky", 80)
        ));
        song.sort(new Comparator<Song>() {
            public int compare(Song o1, Song o2) {
                return o1.getTitle().compareTo(o2.getTitle());
            }
        });


        for (Song s : song) {
            System.out.println(s.getTitle() + "   " + s.getArtist() + "
" + s.getBpm());
        }

    }

}
```

```
public class Song {

        private String title;
        private String artist;
        private int bpm;


        public Song(String title, String artist, int bpm) {
            this.title = title;
```

```
            this.artist = artist;
            this.bpm = bpm;
        }
        public String getTitle() {
            return title;
        }

        public void setTitle(String title) {
            this.title = title;
        }

        public String getArtist() {
            return artist;
        }

        public void setArtist(String artist) {
            this.artist = artist;
        }

        public int getBpm() {
            return bpm;
        }

        public void setBpm(int bpm) {
            this.bpm = bpm;
        }



    }
```



这题学到的东西：

- 匿名内部类：
  正常来说，要实现一个接口，继承一个抽象类时，要新建一个类然后实现（继承）这个接口（抽象类）然后再重写方法，最后在main中new相关的对象才能使用相应的方法。
  而匿名内部类中直接new 接口（抽象类）<T（泛型，传入需要的类型）> {
  @Override
  重写方法 }
  的形式，相当于是简化了上述的步骤（编译后会生成一个相应的.class文件）
  目前好像是只在“方法（）”中当作实参传入时这样用。
- Comparator接口中的compare方法：
  传入两个类型相同的实参，返回一个int。
  只要是看返回的int的正负：
  如果int是正数（以上述代码为例）则o1大于o2（非数字的字符转化为相应的Unicode），则o2在o1之前（默认都是从小到大的）。
  负数的话就o1在o2之前。
  0的话一般是默认顺序，原先在前面的就在前面。
  But，要是o1-o2的数据过大（比如说154984561-（-15645645615152））会导致数据溢出异常。可以用Math.subtractExact，该方法可以判断是否数据溢出，若溢出会抛出

异常：
重写方法改为

```
try{
 return Integer。compare(Math.subtractExact(x,y),0)
 }
catch(ArithmeticException a){
     e.printStackTrace();
}
```

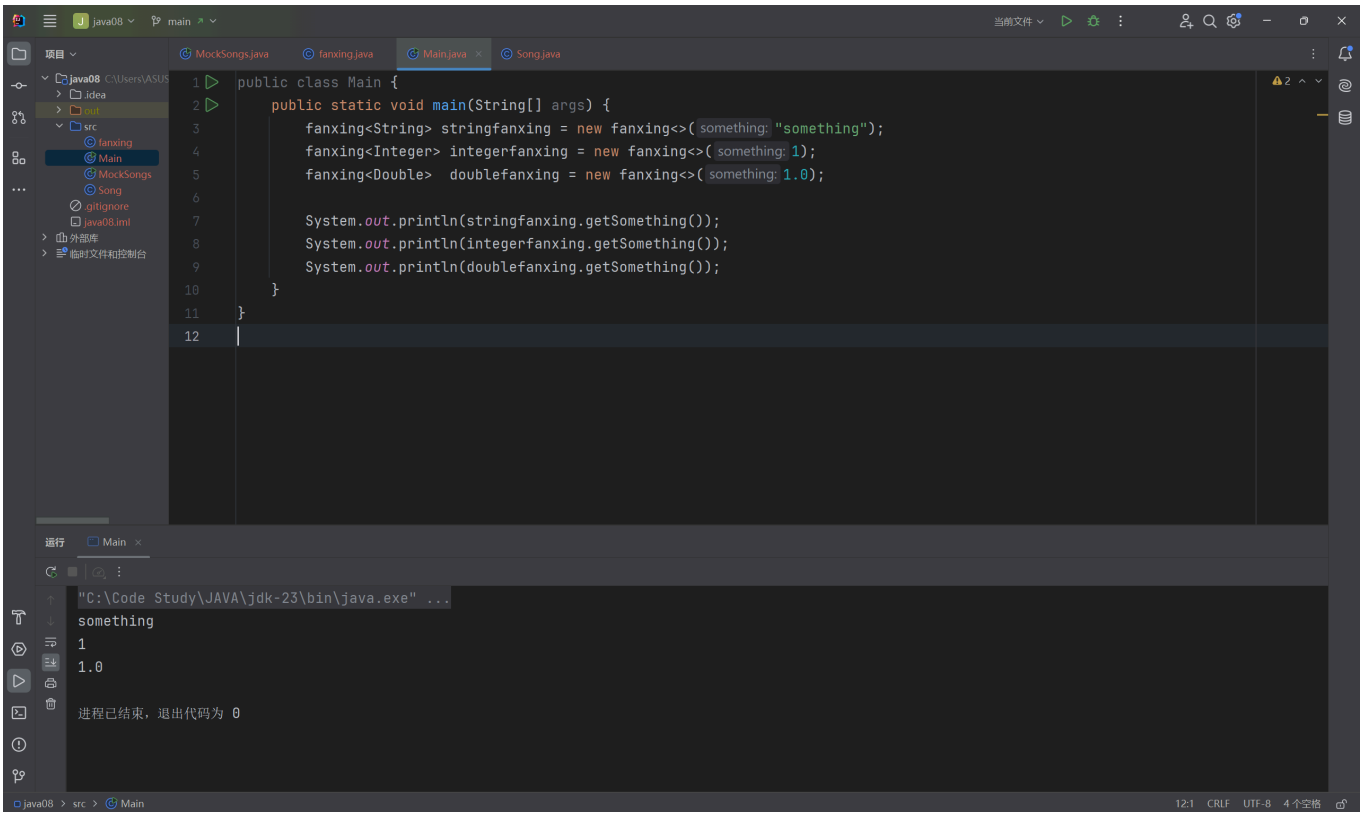
当然在这个题目中我们是字母的unicode的比较，自然不会溢出.....

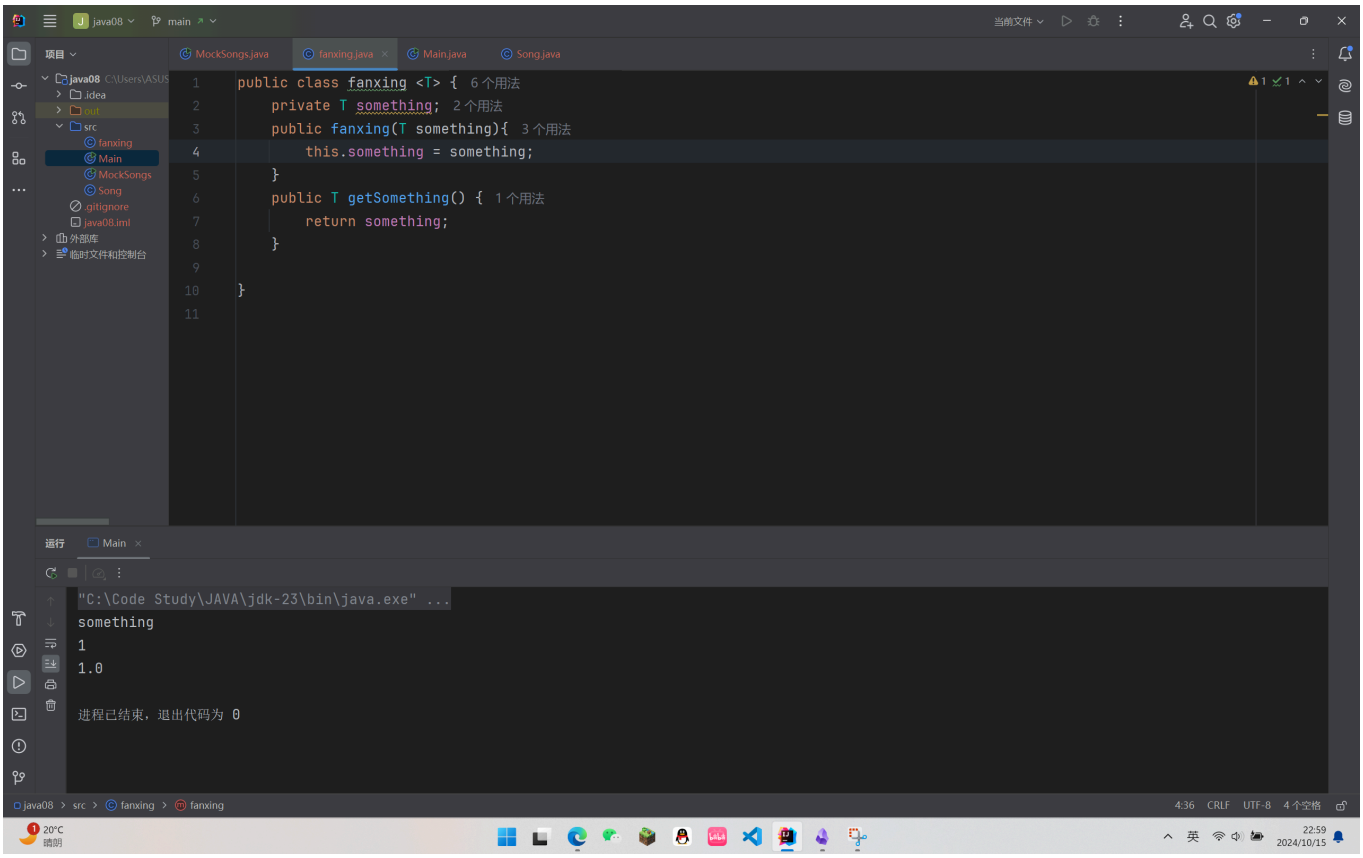
- 增强for循环：

```
for (Song s : song)
```

将List song中的元素一个一个地传入给Song 类型的s变量，然后再用这个s完成操作。
（*要是没弄这个，我大概会用Iterator一个一个赋值了*）

- 泛型：
  （目前我的理解还很肤浅，有错误请直接qq上~~压力我~~）
  泛型是参数化类型。
  假设，我现在有个类，这个类里面有个方法，可以返回你输入的东西。
  那么就有个问题：一个构造方法既可以传入int又可以String又可以double，当然可以利用方法重载多写几个，但是有些麻烦。
  这时候就可以在这个类的类名后面加个<T（只用个T我的markdown会爆红，好奇怪）>，然后构造方法的类型写个（T t）。
  这个时候，在我new对象的时候就可以先规定这里的T是个什么类型，然后这个对象中的T就会全部变为你想要的那个类型：





这就是一个简单的泛型类。

目前我用的最多的是在集合时用泛型规定这个集合存储类是什么，然后就放不进去其他的东西了。（为了安全性和方便）。

然后，这里就不像是new对象时的多态了，这里规定了一个集合是Recreation类型后就不能放入其他的类型，哪怕是其子类Songs。

然后泛型方法：
目前我所知的是：public <T.> T fan(T wuhu){
return wuhu; }
为构造一个泛型方法，如果没有public和返回值T中间的T的话就不是一个泛型方法了。然后这里的T是与前文的类中的T是相互独立的（这个泛型方法中的三个T都是泛型方法里的T不是泛型类的T）。
泛型方法的T是在调用这个方法时确定的。
但是，还没有用到过泛型方法，对它的实践意义还不是很清楚。

- @注释：
  目前来说我只知道它可以说是一种强化版的注释。
  比如在一个方法重写的地方写个@Override，那么就会告诉编译器这里要有一个方法重写，如果没有重写完全，那么编译器就报错不允许运行。
  甚至一些@注释可以自动生成一些简单重复的代码，比如@Data自动生成相应的get，set方法。
  可以简化许多。