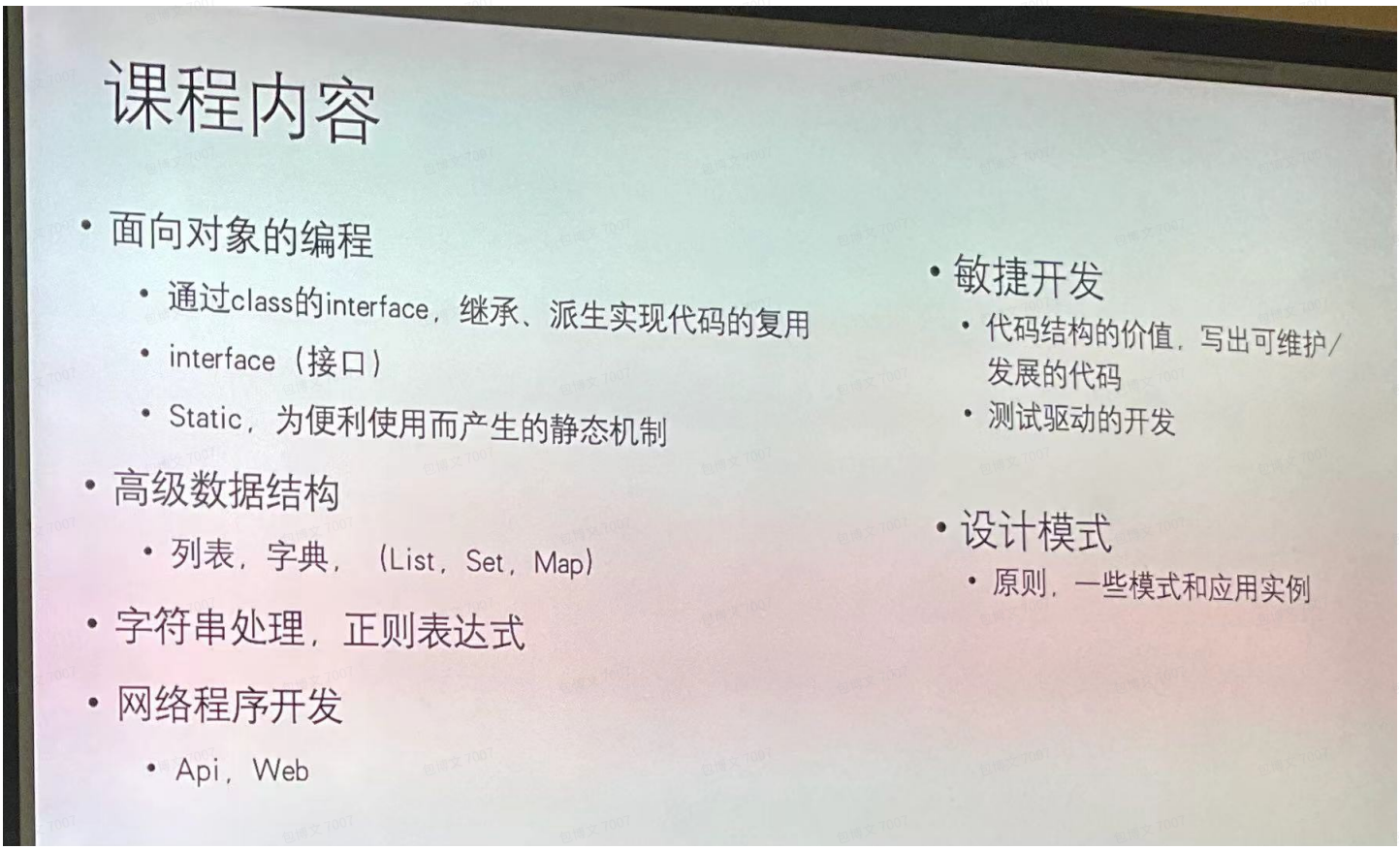
最后一课



面向结构的编程

基本语法

对象和他的接口

- 类：描述具有相同数据元素（属性）和功能（方法）的对象集合。
  - Knows：属性
  - Does：方法
  - 类就是一种自定义数据类型，是一种“会动”的数据结构，既包含数据又包含对这些数据的操作方法。
- 对象

  实例化：通过 `new` 关键字等方式在运行时创建对象。

  接口（interface）与访问：调用对象的方法（public / protected），对外隐藏对象的内部实现（private）。
- 访问控制：

修饰符	类内部	同一个包	不同包的子类	不同包的非子类
private	✔	✘	✘	✘
缺省（default）	✔	✔	✘	✘
protected	✔	✔	✔	✘
public	✔	✔	✔	✔

- **封装**：将数据（属性）和对数据的操作（方法）“包”在一起，对外只提供少量公开接口。
  - 针对size，要通过getSize和setSize方法操作

【例】代码填空

写程序的三步骤

- **Prep code（伪代码）**：先专注逻辑，写出思路，不纠结语法。
- **Test code（测试代码）**：写好测试，用以验证真实代码的正确性。
- **Real code（真实代码）**：在前两步完善后再进行最终实现，提高开发质量。

继承

通过 `extends` 关键字，让子类继承父类的属性和方法。

可以在子类中扩展新方法或重写父类方法（Override）。

```
public class Doctor {
    // 一些共有属性和方法
}

public class FamilyDoctor extends Doctor {
    // 家庭医生特有的属性/方法
}

public class SurgeonDoctor extends Doctor {
    // 外科医生特有的属性/方法
}
```

【例】代码补全

抽象类与抽象方法

- **抽象类**
  - 用 `abstract` 修饰的类，不能直接实例化，只能被子类继承。

- 适用于仅用于提供统一定义和部分实现的父类。

- **抽象方法**
  - 用 abstract 修饰，没有具体方法体（大括号），在子类中必须实现。
  - **意义**：定义子类共有的协议或约定，通过多态用统一的形式调用它们。

Wolf这样的具体类可以被初始化，Animal这样为了共享代码的需要这样初始化：

```java
public abstract class Animal {
    // 可能包含普通方法
    public void breathe() {
        System.out.println("Animal breathes...");
    }

    // 抽象方法，子类必须实现
    public abstract void eat();
}

public class Wolf extends Animal {
    @Override
    public void eat() {
        System.out.println("Wolf eats meat...");
    }
}
```

抽象方法的意义在于定义出一组子类共同的协议

所有子类都有那些抽象的方法，通过多态，可以用统一的形式调用

`Wolf` 是具体类，可以 `new Wolf()`；`Animal` 是抽象类，不可直接实例化。

```java
public abstract void eat();
```

多态

- **定义**：同一个方法调用，根据对象实际所属的子类类型，执行不同的实现。
- **实现**：运行时多态（Dynamic Binding），**父类引用指向子类对象**，通过 `@Override` 机制动态调用子类方法。

```java
// 定义格式：父类类型 变量名=new 子类类型();
Animals animal = new Cat();
```

注意：在使用多态后的父类引用变量调用方法时，会调用子类重写后的方法。

多态中成员的特点

1、多态成员变量：编译运行看左边

```
1   Fu f=new Zi();
2   System.out.println(f.num);//f是Fu中的值，只能取到父中的值
```

2、多态成员方法：编译看左边，运行看右边

```
1   Fu f1=new Zi();
2   System.out.println(f1.show());//f1的门面类型是Fu,但实际类型是Zi,所以调用的是重写后的方法。
```

多态的转型

多态的转型分为向上转型和向下转型两种

向上转型：多态本身就是向上转型过的过程

- 使用格式：父类类型 变量名=new 子类类型();
- 适用场景：当不需要面对子类类型时，通过提高扩展性，或者使用父类的功能就能完成相应的操作。
    - **可以调用父类中定义的方法**：如果子类重写了这些方法，实际调用的是子类的方法（**运行时多态**）。
    - **不能直接调用子类特有的方法**：因为父类引用并不知道子类的具体实现。

向下转型：一个已经向上转型的子类对象可以使用强制类型转换的格式，将父类引用类型转为子类引用各类型

- 使用格式：子类类型 变量名=（子类类型） 父类类型的变量；
- 适用场景：当要使用子类特有功能时。

重写（Override）

重写是指子类定义一个与父类方法 **完全相同** 的方法签名（方法名、参数列表和返回类型），并提供新的实现。

规则：

- 方法名、参数列表和返回类型必须与父类方法一致。

- 访问修饰符不能比父类方法更严格（例如，父类方法是 `protected`，子类方法可以是 `protected` 或 `public`，但不能是 `private`）。
- 可以使用 `@Override` 注解来显式标记重写。

```
class Parent {
    public int change() {
        return 1;
    }
}

class Child extends Parent {
    @Override
    public int change() {
        return 2; // 重写父类方法
    }
}
```

重载（Overload）

重载是指子类定义一个与父类方法 **同名但参数列表不同** 的方法。重载方法可以有不同的返回类型。

规则：

- 方法名必须相同，但参数列表必须不同（参数类型、参数数量或参数顺序）。
- 返回类型可以不同。
- 访问修饰符可以不同。

```
class Parent {
    public int change() {
        return 1;
    }
}

class Child extends Parent {
    // 重载父类方法
    public int change(int x) {
        return x + 1;
    }

    // 重载父类方法，返回类型不同
    public String change(String s) {
        return s + " modified";
    }
```

```
17  }
```

特性	重写（Override）	重载（Overload）
方法签名	必须相同	必须不同
返回类型	必须相同	可以不同
访问修饰符	**不能更严格**	可以不同
作用	修改父类方法的行为	提供同名方法的不同实现

接口

https://blog.csdn.net/sun_shine56/article/details/86621481

接口（`interface`）是 Java 中的一种引用类型，它是**抽象方法和常量值的集合**。接口的核心作用是定义一组规范（方法签名），让实现类去具体实现这些规范。

1. **接口用 `interface` 关键字定义**。
2. **接口中的成员变量默认是 `public static final` 的**（即常量）。
3. **接口中的方法默认是 `public abstract` 的**（即抽象方法）。
4. **接口没有构造方法**，因为接口不能被实例化。
5. **接口支持多继承**，即一个接口可以继承多个接口。
6. **类可以实现多个接口**，从而获得多重继承的效果。
7. 接口可以继承另一个接口，使用 `extends` 关键字。

```
1   public interface Flyable {
2       void fly(); // 抽象方法
3   }
4   public interface Bitable {
5       void bite(); // 抽象方法
6   }
7   public class Bat implements Flyable, Bitable {
8       @Override
9       public void fly() {
10          System.out.println("蝙蝠在飞...");
11      }
12
13      @Override
14      public void bite() {
```

```
15          System.out.println("蝙蝠在咬人...");
16      }
17  }
```

继承和接口的区别

方面	继承	接口
关键字	`extends`	`implements`
关系类型	表示“is-a”关系	表示“can-do”关系
复用性	通过继承父类的实现来复用代码	无具体实现，需由实现类自行提供
多继承	不支持	支持，通过实现多个接口解决单继承局限
实现方式	子类继承父类的方法和字段	类实现接口中的所有抽象方法
访问控制	允许子类访问受保护的父类成员	接口成员默认是 `public`，没有访问控制层级
状态	父类可以有实例变量和方法	接口变量是常量，只能定义行为
设计目的	重用和扩展现有类的功能	定义行为规范，实现松耦合

【例】代码输出

静态

- **静态方法**
  - 不依赖具体对象（实例）即可调用。
    - 静态方法可以直接通过类名调用，而非静态方法需要通过其实例（对象）来调用
  - **限制**：静态方法内无法直接调用非静态属性和方法（因非静态需基于实例存在）。
- **静态变量**
  - 在同一个类的所有实例中**共享**。
  - 类加载时即初始化。
  - 适用于常量或在类层面需要统一管理的属性，如计数器、单例模式中的唯一实例等。
    i. **静态方法访问静态变量**：静态方法只能访问静态变量和其他静态方法。这是因为静态方法在类加载时就存在，而**此时还没有创建类的任何实例**。
    ii. **非静态方法访问静态变量**：非静态方法可以访问静态变量，因为这些方法是在类的实例上调用的，而静态变量是类的一部分。这意味着，无论实例方法在哪个对象上被调用，它们都可以访问类中的静态变量。

iii. **非静态方法访问非静态变量**：非静态方法不仅可以访问静态变量，还可以访问非静态变量，因为这些方法可以操作它们所属对象的状态。

- `this` 关键字可以访问同一个类中的非静态变量（实例变量）。
- `this` 是对当前对象的引用，通过它可以访问当前对象的成员变量和方法。

【例】判断哪些是合法的

- **final**
  - final的变量代表你不能改变它的值
    - `final` 的实例变量必须在声明时或通过构造器初始化
  - final的方法代表你不能override它
  - final的类代表你不能extend（继承）它

特性	final（实例常量）	static final（类常量）
作用范围	属于对象（实例变量）	属于类（类变量）
赋值时机	可以在声明时或构造方法中赋值	必须在声明时或静态代码块中赋值
共享性	每个对象可以有独立的值	所有对象共享同一个值
用途	定义对象的不可变属性	定义全局常量

包装类（Wrapper Classes）

每种基本数据类型（例如 `int`、`double`、`boolean` 等）都有对应的包装类。

包装类是将基本数据类型封装为对象的类，例如：

- `int` -> `Integer`
- `double` -> `Double`
- `boolean` -> `Boolean`

```
1  int x = 32;
2  ArrayList<Integer> list = new ArrayList<>();
3  list.add(x); // 自动装箱
```

自动装箱和拆箱：

- **装箱**：将基本数据类型转换为对应的包装类对象。
- **拆箱**：将包装类对象转换为基本数据类型。

- Java从1.5版本开始支持自动装箱（Autoboxing）和拆箱（Unboxing），简化了使用包装类的代码。

静态导入

Java 5 引入的一种特性，它允许直接使用类的静态成员（如静态方法、静态变量、静态常量）而无需通过类名限定。静态导入可以简化代码，减少重复的类名书写，但过度使用可能会降低代码的可读性。

```
public class Main {
    public static void main(String[] args) {
        double result = Math.sqrt(25); // 需要写类名 Math
        System.out.println(result);
    }
}


import static java.lang.Math.sqrt; // 静态导入 sqrt 方法
public class Main {
    public static void main(String[] args) {
        double result = sqrt(25); // 直接使用 sqrt，无需写类名 Math
        System.out.println(result);
    }
}
```

高级数据结构

String.format

1、格式说明符结构

一个格式说明符的结构如下：

```
%[argument_index$][flags][width][.precision]conversion
```

`%`：标记开始。

`[argument_index$]`：可选，指定参数的位置（第几个参数）。

`[flags]`：可选，控制格式化结果的显示（如是否加逗号、是否左对齐等）。

`[width]`：可选，指定结果的最小宽度。

`[.precision]`：可选，指定浮点数的小数位数或字符串的最大长度。

`conversion`：必要，指定格式化类型（例如整数、浮点数、字符串等）。

2、常用的格式化选项

- **`[flags]` 标志**
  - `,`：添加千位分隔符。

```
String.format("%,d", 1000000); // 输出 "1,000,000"
```

  - `-`：左对齐。

```
String.format("%-10d", 42); // 输出 "42        "
```

  - `0`：在数值前补零。

```
String.format("%010d", 42); // 输出 "0000000042"
```

- **`[width]` 宽度**
  - 设置最小输出宽度，右对齐（默认）。

```
String.format("%6d", 42); // 输出 "    42"
```

- **`[.precision]` 精度**
  - 对于浮点数，表示保留的小数位数。

```
String.format("%.2f", 42.5678); // 输出 "42.57"
```

  - 对于字符串，表示最大字符长度。

```
String.format("%.5s", "HelloWorld"); // 输出 "Hello"
```

3、 多个参数的格式化

- 按参数顺序填充：

```
String.format("The rank is %d out of %.2f", 5, 98.765);
// 输出 "The rank is 5 out of 98.76"
```

- 指定参数顺序：

```
String.format("The first number is %2$d and the second is %1$d", 10, 20);
// 输出 "The first number is 20 and the second is 10"
```

4、示例

- **整数和浮点数**

```
int value = 1000;
double price = 1234.5678;
System.out.println(String.format("%,d", value)); // 输出 "1,000"
System.out.println(String.format("%.2f", price)); // 输出 "1234.57"
```

- **字符和字符串**

```
char c = 'A';
String str = "Hello";
System.out.println(String.format("%c", c)); // 输出 "A"
System.out.println(String.format("%.3s", str)); // 输出 "Hel"
```

- **综合格式化**

```
int rank = 5;
double score = 98.7654;
System.out.println(String.format("Rank: %d, Score: %.2f", rank, score));
// 输出 "Rank: 5, Score: 98.77"
```

Java.util.Calendar

`Calendar` 是 Java 中用于操作日期和时间的抽象类。

它包含了丰富的方法用于日期的计算和格式化。

注意： `Calendar` 是抽象类，不能直接实例化（`new Calendar()` 会报错）。需要使用静态方法 `Calendar.getInstance()` 获取实例。

Collections

ArrayList vs array

```
1   ArrayList<String> myList = new ArrayList<string>();
```

- ArrayList是java api的类，通过add()、remove()来操作元素，还有indexOf()、isEmpty()、size()、max()等
- 其存储的是对象

排序（Set）

我们需要管理一份歌曲列表，列表存储在一个文本文件 `SongList.txt` 中，每行记录一首歌。内容格式如下：

```
1   Pink Moon/Nick Drake
2   Somersault/Zero 7
3   Shiva Moon/Prem Joshua
4   Circles/BT
5   Deep Channel/Afro Celts
6   Passenger/Headmix
7   Listen/Tahiti 80
```

每行的格式是：**歌曲名/歌手名**

我们的目标：

1. 读取这些数据并存储在程序中。
2. 将歌曲按照歌曲名排序。
3. 后续可以扩展为按歌手名排序。

初步解决方法：使用 `ArrayList<String>` 存储和排序

我们可以用 `ArrayList<String>` 来存储每一首歌的名字。步骤如下：

1. 使用文件读取工具 `BufferedReader` 读取文件内容。
2. 将每行按 `/` 分割，提取歌曲名，存入 `ArrayList`。
3. 使用 `Collections.sort()` 对 `ArrayList` 进行排序。

```
import java.io.*;
import java.util.*;

public class Jukebox {
    ArrayList<String> songList = new ArrayList<>();

    public static void main(String[] args) {
        new Jukebox().go();
    }

    public void go() {
        getSongs();
        System.out.println("Original List: " + songList);
        Collections.sort(songList); // 按字母顺序排序
        System.out.println("Sorted List: " + songList);
    }

    void getSongs() {
        try {
            File file = new File("SongList.txt");
            BufferedReader reader = new BufferedReader(new FileReader(file));
            String line;
            while ((line = reader.readLine()) != null) {
                addSong(line);
            }
            reader.close();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    void addSong(String lineToParse) {
        String[] tokens = lineToParse.split("/"); // 按斜线分割
        songList.add(tokens[0]); // 只存歌曲名
    }
}
```

`Collections.sort` 是 Java 中用于对集合进行排序的静态方法，其有两种重载版本：

1. `Collections.sort(List<T> list)`
   - 泛型 `T` 必须实现 `Comparable<? super T>` 接口。
   - 例如，如果 `T` 是 `Integer`，则 `Integer` 实现了 `Comparable<Integer>`。
2. `Collections.sort(List<T> list, Comparator<? super T> c)`
   - 泛型 `T` 可以是任何类型。
   - `Comparator<? super T>` 表示比较器可以接受 `T` 或其父类型。

困难1：只能存储歌曲名，信息不够全面

初步实现中，我们只存储了歌曲名，这样有几个问题：

- 无法存储更复杂的信息（如歌手、评分等）。
- 对歌曲排序时，固定按名字排序，不能按歌手或其他属性排序。

修改1：引入 `Song` 类

我们将歌曲的数据封装到一个 `Song` 类中，使每首歌曲可以包含多个属性，例如：

- `title`：歌曲名
- `artist`：歌手
- `rating`：评分
- `bpm`：节奏（每分钟拍数）

```
1   public class Song {
2       private String title;
3       private String artist;
4       private String rating;
5       private String bpm;
6
7       // 构造函数
8       public Song(String title, String artist, String rating, String bpm) {
9           this.title = title;
10          this.artist = artist;
11          this.rating = rating;
12          this.bpm = bpm;
13      }
14
15      // Getter方法
16      public String getTitle() {
17          return title;
18      }
```

```
19
20      public String getArtist() {
21          return artist;
22      }
23
24      public String getRating() {
25          return rating;
26      }
27
28      public String getBpm() {
29          return bpm;
30      }
31
32      // 重写toString()方法，便于打印
33      @Override
34      public String toString() {
35          return title;
36      }
37  }
```

为什么重写 `toString` 方法？ 默认的 `toString` 方法会打印对象的内存地址，而不是我们希望的有意义的信息。通过重写 `toString` 方法，我们可以定义打印对象时显示的内容，比如歌曲名。

修改2：使用 `ArrayList<Song>`

将原来的 `ArrayList<String>` 改为 `ArrayList<Song>` ，并修改 `addSong` 方法，解析出所有属性并创建 `Song` 对象。

```
1   import java.io.*;
2   import java.util.*;
3
4   public class Jukebox {
5       ArrayList<Song> songList = new ArrayList<>();
6
7       public static void main(String[] args) {
8           new Jukebox().go();
9       }
10
11      public void go() {
12          getSongs();
13          System.out.println("Original List: " + songList);
14          Collections.sort(songList); // 尝试排序
15          System.out.println("Sorted List: " + songList);
16      }
17
```

```
    void getSongs() {
        try {
            File file = new File("SongList.txt");
            BufferedReader reader = new BufferedReader(new FileReader(file));
            String line;
            while ((line = reader.readLine()) != null) {
                addSong(line);
            }
            reader.close();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    void addSong(String lineToParse) {
        String[] tokens = lineToParse.split("/"); // 按斜线分割
        Song nextSong = new Song(tokens[0], tokens[1], "5", "120"); // 假设评分和bpm为默认值
        songList.add(nextSong);
    }
}
```

困难2：`Collections.sort(songList)` 无法运行

原因是 `Song` 类未实现 `Comparable` 接口

修改1：实现 `Comparable` 接口

`Comparable` 接口允许类的对象进行排序。实现 `Comparable` 接口的类必须重写 `compareTo` 方法，定义排序规则。

`compareTo` 方法的返回值规则：

- **负整数**：当前对象小于指定对象（当前对象排在前面）。
- **零**：当前对象等于指定对象。
- **正整数**：当前对象大于指定对象（当前对象排在后面）。

排序规则	compareTo 实现	解释
从小到大	return this.age - other.age;	当前对象小于指定对象时返回负整数
从大到小	return other.age - this.age;	当前对象大于指定对象时返回负整数

```
public class Song implements Comparable<Song> {
    private String title;
```

```
    private String artist;
    private String rating;
    private String bpm;

    public Song(String title, String artist, String rating, String bpm) {
        this.title = title;
        this.artist = artist;
        this.rating = rating;
        this.bpm = bpm;
    }

    public String getTitle() {
        return title;
    }

    @Override
    public String toString() {
        return title;
    }

    // 实现compareTo方法，按标题排序
    @Override
    public int compareTo(Song other) {
        return title.compareTo(other.getTitle());
    }
}
```

修改2：扩展功能

为了支持按其他属性排序，我们可以使用 `Comparator` 接口定义自定义排序规则。`Comparator` 接口是 Java 中用于定义自定义排序规则的接口。它包含多个方法，其中最核心的是 `compare` 方法，但 Java 8 之后还引入了许多默认方法和静态方法，增强了 `Comparator` 的功能。

- 自定义Comparator

```
import java.util.*;

public class ArtistCompare implements Comparator<Song> {
    @Override
    public int compare(Song one, Song two) {
        return one.getArtist().compareTo(two.getArtist());
    }
}

```

```
10  Comparator<Person> reversedComparator = ageComparator.reversed(); // 按年龄从大到小排序
```

- 例：按歌手排序

```
1   public void go() {
2       getSongs();
3       System.out.println("Original List: " + songList);
4
5       // 按标题排序
6       Collections.sort(songList);
7       System.out.println("Sorted by Title: " + songList);
8
9       // 按歌手排序
10      ArtistCompare artistCompare = new ArtistCompare();
11      Collections.sort(songList, artistCompare);
12      System.out.println("Sorted by Artist: " + songList);
13  }
```

特性	Comparable 接口	Comparator 接口
接口定义	java.lang.Comparable<T>	java.util.Comparator<T>
实现方式	需要修改类的源代码，实现 Comparable 接口	不需要修改类的源代码，独立实现 Compa 口
方法名	compareTo(T o)	compare(T o1, T o2)
方法参数	接受一个参数（与当前对象比较）	接受两个参数（比较两个对象）
返回值	负数：当前对象 < 传入对象 零：当前对象 = 传入对象 正数：当前对象 > 传入对象	负数：第一个对象 < 第二个对象 零：两个对象相等 正数：第一个对象 > 第二个对象
排序逻辑	定义对象的自然排序顺序	定义对象的自定义排序顺序
使用场景	当对象有唯一的自然排序顺序时	当需要多种排序方式或无法修改类源代码
调用方式	Collections.sort(list)	Collections.sort(list, comparator)
灵活性	较低，只能定义一种排序方式	较高，可以定义多种排序方式
是否侵入性	是，需要修改类的源代码	否，不需要修改类的源代码
适用场景示例	学生按年龄自然排序	学生按年龄、姓名等多种方式排序
线程安全性	取决于实现	取决于实现
Java 版本支持	从 Java 1.2 开始支持	从 Java 1.2 开始支持
是否支持链式比较	不支持	支持，可以使用 Comparator.thenComparir 链式比较
是否支持逆序	不支持	支持，可以使用 Comparator.reversed() 实

Comparable

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

class Student implements Comparable<Student> {
    String name;
    int age;

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

```
13
14      @Override
15      public int compareTo(Student other) {
16          // 按年龄排序
17          return this.age - other.age;
18      }
19
20      @Override
21      public String toString() {
22          return name + " (" + age + ")";
23      }
24  }
25
26  public class ComparableExample {
27      public static void main(String[] args) {
28          List<Student> students = new ArrayList<>();
29          students.add(new Student("Alice", 20));
30          students.add(new Student("Bob", 18));
31          students.add(new Student("Charlie", 22));
32
33          Collections.sort(students); // 使用 Comparable 排序
34          System.out.println("Sorted by age (Comparable): " + students);
35      }
36  }
```

Comparator

```
1   import java.util.ArrayList;
2   import java.util.Collections;
3   import java.util.Comparator;
4   import java.util.List;
5
6   class Student {
7       String name;
8       int age;
9
10      public Student(String name, int age) {
11          this.name = name;
12          this.age = age;
13      }
14
15      @Override
16      public String toString() {
17          return name + " (" + age + ")";
18      }
```

```java
}

class NameComparator implements Comparator<Student> {
    @Override
    public int compare(Student s1, Student s2) {
        // 按姓名排序
        return s1.name.compareTo(s2.name);
    }
}

public class ComparatorExample {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        students.add(new Student("Alice", 20));
        students.add(new Student("Bob", 18));
        students.add(new Student("Charlie", 22));

        Collections.sort(students, new NameComparator()); // 使用 Comparator 排序
        System.out.println("Sorted by name (Comparator): " + students);
    }
}
```

困难3：存在重复行

```
Pink Moon/Nick Drake/5/80
Listen/Tahiti 80/5/90
Listen/Tahiti 80/5/90
Circles/BT/5/110
Circles/BT/5/110
```

List，Set，Map

特性	List（列表）	Set（集合）	Map（映射）
是否允许重复	允许	不允许	键不允许重复，值
是否有序	有序（按插入顺序）	无序（除非使用 LinkedHashSet 或 TreeSet）	无序（除非使用 Li 或 TreeMap）
存储方式	单元素集合	单元素集合	键值对集合
实现类	ArrayList, LinkedList, Vector	HashSet, LinkedHashSet, TreeSet	HashMap, LinkedHa , Hashtable
适用场景	需要维护顺序或允许重复元素	需要存储唯一元素	需要通过键快速查
派生自	Collection 接口	Collection 接口	独立的 Map 接口

修改1：使用 `Set` 来去重

使用 `Set` 接口实现类（如 `HashSet`）存储歌曲数据，`Set` 的特性是**不允许重复元素**。

在添加数据时，`Set` 会自动检查是否已经存在相同的元素。

因此我们修改程序，将 `ArrayList<Song>` 替换为 `HashSet<Song>`：

```
1  import java.io.*;
2  import java.util.*;
3
4  public class Jukebox {
5      HashSet<Song> songSet = new HashSet<>();
6
7      public static void main(String[] args) {
8          new Jukebox().go();
9      }
10
11     public void go() {
12         getSongs();
13         System.out.println("Unique Songs: " + songSet);
14     }
15
16     void getSongs() {
17         try {
18             File file = new File("SongListMore.txt");
19             BufferedReader reader = new BufferedReader(new FileReader(file));
20             String line;
21             while ((line = reader.readLine()) != null) {
22                 addSong(line);
23             }
24             reader.close();
```

```
25            } catch (Exception ex) {
26                ex.printStackTrace();
27            }
28        }
29
30        void addSong(String lineToParse) {
31            String[] tokens = lineToParse.split("/");
32            Song nextSong = new Song(tokens[0], tokens[1], tokens[2], tokens[3]);
33            songSet.add(nextSong); // 自动去重
34        }
35    }
```

困难3.1：如何判断对象是否重复？

`Set` 的重复判断依据是对象的 `hashCode()` 值和 `equals()` 方法。

如果没有正确实现这两个方法， `Set` 可能无法识别两个逻辑上“相同”的对象。

原先的 `hashCode()` 是用地址计算，所以哪怕相同 `title` 的 `hashcode` 也是不同的。

修改1.1：重写 `hashCode()` 和 `equals()` 方法

思路：

1. **`hashCode()` 方法**：
   - 生成对象的哈希值，用于快速比较。
   - 现在用属性计算，所以就可以用它来建表了。
   - 以前相同元素存的地址不同，hashcode就不一样，就没法去重。
2. **`equals()` 方法**：
   - 比较两个对象是否相等。
   - 如果两个对象的 `hashCode()` 相等，则 `Set` 会进一步调用 `equals()` 来确定是否相等。

```
1   public class Song implements Comparable<Song> {
2       private String title;
3       private String artist;
4       private String rating;
5       private String bpm;
6
7       public Song(String title, String artist, String rating, String bpm) {
8           this.title = title;
9           this.artist = artist;
10          this.rating = rating;
11          this.bpm = bpm;
12      }
```

```
13
14      public String getTitle() {
15          return title;
16      }
17
18      @Override
19      public String toString() {
20          return title;
21      }
22
23      // 重写equals方法
24      @Override
25      public boolean equals(Object aSong) {
26          if (this == aSong) return true;
27          if (aSong == null || getClass() != aSong.getClass()) return false;
28          Song song = (Song) aSong;
29          return title.equals(song.title);
30      }
31
32      // 重写hashCode方法
33      @Override
34      public int hashCode() {
35          return title.hashCode(); // 使用title的hashCode作为Song的hashCode
36      }
37
38      @Override
39      public int compareTo(Song other) {
40          return title.compareTo(other.getTitle());
41      }
42  }
```

修改2：使用 `TreeSet` 保持有序

`TreeSet` 是一种基于红黑树的集合，它的主要特性包括：

- **自动排序**：`TreeSet` 会根据元素的“自然顺序”（`Comparable`）或“自定义顺序”（`Comparator`）进行排序。
- **去重**：`TreeSet` 和 `HashSet` 一样，不允许重复元素。
- **效率**：常见操作（如添加、删除）时间复杂度为 O(logn)。

```
1   public void go() {
2       getSongs();
3       System.out.println("Original List: " + songList);
4
```

```
        // 使用 Collections.sort 进行排序
        Collections.sort(songList);
        System.out.println("Sorted List: " + songList);

        // 使用 TreeSet 保持有序去重
        TreeSet<Song> songSet = new TreeSet<>();
        songSet.addAll(songList);
        System.out.println("TreeSet (Sorted & Unique): " + songSet);
    }
```

TreeSet 的优点和缺点

- **优点**：
  a. 自动排序，不需要额外调用 `Collections.sort`。
  b. 避免重复元素，无需手动检查。
- **缺点**：
  a. 不适合需要频繁插入和删除的场景，性能不如 `HashSet`。
  b. 如果不需要顺序，`TreeSet` 会带来不必要的性能开销。

问题3.2：`TreeSet` 中未实现 `Comparable` 的问题

如果 `TreeSet` 的元素类未实现 `Comparable`，或者排序逻辑需要自定义，编译会报错。

```
TreeSet<Book> tree = new TreeSet<>();
tree.add(new Book("How Cats Work")); // 错误：Book 未实现 Comparable
```

实现 `Comparable` 接口：在 `Book` 类中实现 `Comparable` 接口，并定义 `compareTo` 方法。

```
class Book implements Comparable<Book> {
    String title;

    public Book(String t) {
        title = t;
    }

    @Override
    public int compareTo(Book b) {
        return title.compareTo(b.title);
    }
}
```

使用自定义 `Comparator`： 如果不希望修改类本身，可以在创建 `TreeSet` 时传入自定义比较器：

```
TreeSet<Book> tree = new TreeSet<>(Comparator.comparing(b -> b.title));
```

Map

`Map` 是一种键值对结构，常用实现有 `HashMap`、`TreeMap` 和 `LinkedHashMap`。常见场景包括：

- 通过键快速查找值。
- 保证键的唯一性。

```
public class TestMap {
    public static void main(String[] args) {
        HashMap<String, Integer> scores = new HashMap<>();
        scores.put("Kathy", 42);
        scores.put("Bert", 343);
        scores.put("Skyler", 420);

        System.out.println(scores);
        System.out.println("Bert's score: " + scores.get("Bert"));
    }
}

// 输出
// {Kathy=42, Skyler=420, Bert=343}
// Bert's score: 343
```

泛型

写法	解释	特点
ArrayList<Animal>	存储 Animal 类型的对象	只能存储 Animal 及其子类，读取时
ArrayList<? extends Animal>	存储 Animal 或其子类的对象	只能读取，不能添加（除了 null）
ArrayList<? super Animal>	存储 Animal 或其父类的对象	可以添加 Animal 及其子类，读取时
ArrayList<?>	存储任意类型的对象	只能读取，不能添加（除了 null）
ArrayList<T>	泛型类型参数，T 是占位符	需要在类、接口或方法中定义，使用 类型
ArrayList<ArrayList<T>>	存储 ArrayList，内部 ArrayList 的元素类型为 T	用于处理多维集合
ArrayList<T extends Animal>	T 必须是 Animal 或其子类	限制泛型类型的范围

1、为什么数组不适合作为泛型的替代品？

- **数组的类型检查**：运行时检查。
- **泛型的类型检查**：编译时检查，更加安全。

```
Object[] array = new String[10];
array[0] = 42; // 编译通过，运行时抛出 ArrayStoreException
```

2、使用泛型的优势

- 类型安全：在编译时捕获错误。
- 可读性更强：明确集合中存储的元素类型。

```
public void go() {
    ArrayList<Animal> animals = new ArrayList<Animal>();
    animals.add(new Dog());
    animals.add(new Cat());
    animals.add(new Dog());
    takeAnimals(animals);

    ArrayList<Dog> dogs = new ArrayList<Dog>();
    dogs.add(new Dog());
    dogs.add(new Dog());
    takeAnimals(dogs);
}

public void takeAnimals(ArrayList<Animal> animals) {
    for (Animal a : animals) {
        a.eat();
```

```
    }
}
```

- 协边通配符修正

```
public void takeAnimals(ArrayList<? extends Animal> animals) {
    for (Animal a : animals) {
        a.eat();
    }
}

```

`? extends Animal` 表示可以接收 `Animal` 或其子类的泛型集合，适合只读操作（如遍历）。

- 逆变通配符修正

```
public void takeAnimals(ArrayList<? super Animal> animals) {
    for (Object obj : animals) {
        ((Animal) obj).eat();
    }
}

```

`? super Animal` 表示可以接收 `Animal` 或其父类的泛型集合，适合写操作（如添加对象）。

字符串处理/正则

字符类 Character

- JAVA包装char得到的类

```
Character ch = new Character('a');
Character ch = 'a';         // 自动装箱 autoboxing
```

- **Character方法：**

SN	方法描述
1	isLetter() 确定具体的char值是一个字母
2	isDigit() 确定具体的char值是一个数字
3	isWhitespace() 确定具体的char值是一个空格
4	isUpperCase() 确定具体的char值是一个大写字母
5	isLowerCase() 确定具体的char值是一个小写字母
6	toUpperCase() 返回指定字符值的大写形式
7	toLowerCase() 返回指定字符值的小写写形式
8	toString() 返回代表指定的字符值的一个String对象,即一个字符的字符串

- 转义字符

转义序列	描述
\t	在文本中插入一个制表符。
\b	在文本中插入一个退格。
\n	在文本中插入一个换行符。
\r	在文本中插入一个回车。
\f	在文本中插入一个换页。
\'	在文本中插入一个单引号字符。
\\	在文本中插入一个反斜杠字符。

字符串类 String

1、final：不可被继承，不可被修改

- 任何对 `String` 的修改操作（如拼接、替换等）都会创建一个新的 `String` 对象，而不是修改原有的对象。
- 如果需要频繁修改字符串，Java 提供了可变的字符串类：
  i. `StringBuilder`：非线程安全，性能较高。
  ii. `StringBuffer`：线程安全，性能稍低。

```
1   StringBuilder sb = new StringBuilder("Hello");
```

```
sb.append(" World"); // 直接修改 StringBuilder 对象
String result = sb.toString(); // 转换为 String
```

2、创建字符串

- 直接使用字符串字面量

```
String greeting = "Hello world!";
```

- 使用字符数组创建字符串

```
char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };
String helloString = new String(helloArray);
```

- 使用 String.format 格式化创建字符串

```
String name = "Alice";
int age = 25;
String message = String.format("My name is %s and I am %d years old.", name, age);
// message 的值为 "My name is Alice and I am 25 years old."
```

3、字符串查找

- `indexOf(String s)`：
  - 返回参数字符串 `s` 在指定字符串中首次出现的索引位置。
  - 如果未找到，返回 `-1`。

```
String str = "We are students";
int ssr = str.indexOf("a"); // ssr 的值为 3
```

- `lastIndexOf(String str)`：
  - 返回字符串 `str` 最后一次出现的索引位置。
  - 如果未找到，返回 `-1`。
  - 如果参数是空字符串 `""`，则返回结果与 `length()` 方法相同。

```java
public class Main {
    public static void main(String[] args) {
        String text = "Hello, World. Hello again!";
        int index = text.lastIndexOf("Hello");
        System.out.println("Last index of 'Hello': " + index); // 输出 17
    }
}
```

4、获取指定索引处的字符

- `charAt(int index)`：
  - 返回指定索引处的字符。

```java
char ch = str.charAt(1); // 获取第二个字符
```

5、截取字符串

- `substring(int beginIndex)`：
  - 返回从指定索引位置开始到字符串结尾的子串。

```java
String sub = str.substring(3); // 返回 "are students"
```

- `substring(int beginIndex, int endIndex)`：
  - 返回从 `beginIndex` 开始到 `endIndex` 之前的子串。

```java
String sub = str.substring(3, 6); // 返回 "are"
```

6、去除空格：trim()

- 返回字符串的副本，忽略前导空格和尾部空格。

```java
String trimmed = "  Hello  ".trim(); // 返回 "Hello"
```

7、替换字符串

- `replace(char oldChar, char newChar)`：

- 将指定的字符或字符串替换为新的字符或字符串。

```
String replaced = str.replace('e', 'E'); // 替换所有 'e' 为 'E'
```

8、判断字符串的开始和结束

`startsWith(String prefix)`：

- 判断字符串是否以指定的前缀开始。

```
boolean starts = str.startsWith("We"); // 返回 true
```

- `endsWith(String suffix)`：
  - 判断字符串是否以指定的后缀结束。

```
boolean ends = str.endsWith("stdent"); // 返回 false
```

9、比较字符串

- `equals(String otherstr)`：
  - 比较两个字符串的字符和长度，区分大小写。

```
boolean isEqual = str.equals("we are students"); // 返回 true
```

- `equalsIgnoreCase(String otherstr)`：
  - 比较两个字符串的字符和长度，忽略大小写。

```
boolean isEqual = str.equalsIgnoreCase("we are students"); // 返回 true
```

- `compareTo(String anotherString)`：
  - 按字典顺序比较两个字符串，基于字符的 Unicode 值。
  - 返回负整数、正整数或 0。

```
```

```java
public class Main {
    public static void main(String[] args) {
        String str1 = "apple";
        String str2 = "banana";
        String str3 = "apple";

        int comparison1 = str1.compareTo(str2);
        int comparison2 = str1.compareTo(str3);

        System.out.println("Comparison between 'apple' and 'banana': " + comparison1); // 输出负数
        System.out.println("Comparison between 'apple' and 'apple': " + comparison2); // 输出0
    }
}
```

10、大小写转换

`toLowerCase()`

```java
String lower = str.toLowerCase(); // 返回 "we are students"
```

`toUpperCase()`

```java
String upper = str.toUpperCase(); // 返回 "WE ARE STUDENTS"
```

11、分割字符串

- `split(String regex)`：
  - 按指定的分隔符或正则表达式分割字符串，结果存放在字符数组中。

```java
String str = "Hello World";
String[] parts = str.split(" ");
System.out.println(Arrays.toString(parts)); // 输出: [Hello, World]

String data = "apple,banana=cherry";
String[] parts = data.split(",|=");
System.out.println(Arrays.toString(parts)); // 输出: [apple, banana, cherry]
```

正则表达式

元字符

- \b：代表着单词的开头或结尾，也就是单词的分界处, 它只匹配一个位置
- .：匹配除了换行符以外的任意字符
- *：指定*前边的内容可以连续重复使用任意次（≥0）以使整个表达式得到匹配

```
\bhi\b.*\bLucy\b
//先是一个单词hi，然后是任意个任意字符(但不能是换行)，最后是Lucy这个单词
```

- \d：匹配一位数字(0，或1，或2，或……)
  - d{2}：这里\d后面的{2}({8})的意思是前面\d必须连续重复匹配2次(8次)
  - \d+：匹配1个或更多连续的数字（≥1）

```
0\d{2}-\d{8}
// 中国的电话号码。当然，这个例子只能匹配区号为3位的情形
```

- ^：匹配字符串的开始
- $：匹配字符串的结束

```
^\w+
// 匹配一行的第一个单词(或整个字符串的第一个单词)
```

- \s：匹配任意的空白符，包括空格，制表符(Tab)，换行符，中文全角空格等
- \w：匹配字母或数字或下划线或汉字等
  - \w{6}：刚好6个字符
  - \w{6,}：匹配6次或更多次
  - \w{5,12}：匹配5到12次

```
\ba\w*\b
// 匹配以字母a开头的单词，如abc111
```

转义字符

元字符本身无法被指定，所以就得使用\来取消这些字符的特殊意义

```
deerchao\.cn
//匹配deerchao.cn
C:\\Windows
// 匹配C:\Windows
```

自定义字符集合

- 使用中括号扩住即可

```
[aeiou]
// 匹配任何一个英文元音字母
[.?!]
// 匹配标点符号(.或?或!)。
```

- 也可在中括号中使用'-'表示范围

```
[0-9]，[a-z0-9A-Z_]也完全等同于\w（如果只考虑英文的话）
```

分支条件

```
\(? 0 \d{2} [) -]? \d{8}
```

这个表达式可以匹配以下几种格式的电话号码：

- `(010)88886666`
- `022-22334455`
- `02912345678`

也会匹配一些不正确的格式，比如：

- `010)12345678`：右括号没有对应的左括号。
- `(022-87654321`：左括号没有对应的右括号。

```
\( 0 \d{2} \) [- ]? \d{8} | 0 \d{2} [- ]? \d{8}
```

分组

如果你想重复多个字符（而不仅仅是一个字符），可以使用小括号 `()` 将这些字符包裹起来，形成一个**分组**（也称为子表达式）。然后，你可以在分组后面加上限定符来指定重复次数。

```
(\d{1,3}\.){3}      \d{1,3}
```

这个表达式可以匹配类似以下格式的IP地址：

- `192.168.1.1`
- `10.0.0.1`
- `255.255.255.0`

也会匹配一些无效的IP地址，比如：

- `256.300.888.999`：IP地址的每个部分（称为“段”）的取值范围是0到255，而这里的某些段超过了255。

```
((   2[0-4]\d   | 25[0-5] | [01]?  \d \d?   )  \.  ){3}     (2[0-4]\d|25[0-5]|[01]?\d\d?)
```

反义

反义用于匹配**不属于某个字符类**的字符。换句话说，它是对某个字符类的否定。通过反义，我们可以匹配那些不符合特定规则的字符。

在正则表达式中，反义通常通过以下方式实现：

- 使用 `\` 开头的反义代码（如 `\W`、`\S`、`\D` 等）。
- 使用 `[^...]` 形式的否定字符类。

代码/语法	说明
\W	匹配任意**不是**字母、数字、下划线、汉字的字符。等价于 [^a-zA-Z0-9_]。
\S	匹配任意**不是**空白符的字符。空白符包括空格、制表符（\t）、换行符（\n）等。
\D	匹配任意**非数字**的字符。等价于 [^0-9]。
\B	匹配**不是**单词开头或结束的位置。
[^x]	匹配除了 x 以外的任意字符。
[^aeiou]	匹配除了 a、e、i、o、u 这几个字母以外的任意字符。

```
< a [^>]+ >
字符串：<a href="https://example.com">Link</a>
匹配结果：<a href="https://example.com">。
```

后向引用

后向引用（Backreference）是指在正则表达式中，引用前面已经捕获的分组（子表达式）所匹配的文本。通过后向引用，我们可以重复使用前面分组匹配的内容，从而实现更复杂的匹配规则。

- 分组是通过小括号 `()` 定义的。
- 每个分组会自动分配一个组号，从左到右，第一个分组的组号为 `1`，第二个为 `2`，以此类推。
- 后向引用使用 `\1`，`\2`，`\3` 等语法来引用对应分组的内容。

零宽断言

零宽断言用于匹配一个位置，这个位置必须满足特定的条件（断言），但它不会消耗字符（即不会匹配具体的字符内容）。零宽断言分为两种：

- **正向零宽断言**：断言某个条件必须成立。
- **负向零宽断言**：断言某个条件必须不成立。

1、 零宽度正预测先行断言（`(?=exp)`）

- **语法**：`(?=exp)`
- **作用**：断言当前位置的**后面**必须匹配表达式 `exp`，但 `exp` 本身不会被包含在匹配结果中。
- **特点**：只匹配位置，不消耗字符。

【例】匹配以 `ing` 结尾的单词的前半部分

```
\b\w+(?=ing\b)
在 I'm singing while you're dancing. 中，匹配 sing 和 danc。
```

2、零宽度正回顾后发断言（`(?<=exp)`）

- **语法**：`(?<=exp)`
- **作用**：断言当前位置的**前面**必须匹配表达式 `exp`，但 `exp` 本身不会被包含在匹配结果中。
- **特点**：只匹配位置，不消耗字符。

【例】匹配以 `re` 开头的单词的后半部分

```
(?<=\bre)\w+\b
reading 中，ading 前面是 re，因此匹配 ading。
```

3、零宽度负预测先行断言（`(?!exp)`）

- **语法**：`(?!exp)`
- **作用**：断言当前位置的**后面**不能匹配表达式 `exp`。
- **特点**：只匹配位置，不消耗字符。

【例】匹配三位数字，且后面不能是数字

```
\d{3}(?!\d)
匹配的文本：
123（如果后面不是数字）。
456（如果后面不是数字）。
不匹配的文本：
1234（因为 123 后面是数字 4）。
```

4、零宽度负回顾后发断言（`(?<!exp)`）

- **语法**：`(?<!exp)`
- **作用**：断言当前位置的**前面**不能匹配表达式 `exp`。
- **特点**：只匹配位置，不消耗字符。

【例】匹配前面不是小写字母的七位数字

```
(?<![a-z])\d{7}
匹配的文本：
1234567（如果前面不是小写字母）。
A1234567（如果前面不是小写字母）。
不匹配的文本：
a1234567（因为前面是小写字母 a）。
```

贪婪与懒惰

- **贪婪匹配（Greedy Matching）**

默认情况下，正则表达式中的限定符（如 `*`，`+`，`?`，`{n,m}` 等）是**贪婪**的。这意味着它们会尽可能多地匹配字符，直到无法匹配为止。

```
a.*b
在 aabab 中，匹配整个字符串 aabab。

a.*?b
```

```
5    在 aabab 中，匹配字符串 aab 和 ab。
```

- **懒惰匹配（Lazy Matching）**

懒惰匹配（也称为非贪婪匹配）是指在匹配时尽可能少地匹配字符。可以通过在限定符后面加上 `?` 来实现懒惰匹配。

代码/语法	说明
*?	重复任意次，但尽可能少重复
+?	重复1次或更多次，但尽可能少重复
??	重复0次或1次，但尽可能少重复
{n,m}?	重复n到m次，但尽可能少重复
{n,}?	重复n次以上，但尽可能少重复

1. 长度为8-10的用户密码（以字母开头、数字、下划线）

```
^[a-zA-Z]\w{7,10}$
```

2. 电子邮箱验证：

```
^\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*$`
```

3. URL地址验证：

```
^http://([\w-]+\.)+[\w-]+(/[\w-./?%&=]*)?$
```

4. 电话号码的验证：

```
 0\d{3}-\d{7}|0\d{2}-\d{8}
```

5. 简单的身份证号验证：

```
 \d{15}|\d{18}$
```

设计模式

在软件开发中，某些问题会反复出现。例如：

- 如何确保一个类只有一个实例？（单例模式）
- 如何动态地为对象添加功能？（装饰模式）
- 如何解耦发送者和接收者？（命令模式）

设计模式提供了这些问题的标准化解决方案，避免了开发者重复“造轮子”。

创建型模式

创建对象时，需要动态地决定怎样创建对象，创建哪些对象，以及如何组合和表示这些对象。

1、单例模式

意图：保证一个类仅有一个实例，并提供一个访问它的全局访问点。

主要解决：一个全局使用的类频繁地创建与销毁。

注意：

1、单例类只能有一个实例。

2、单例类必须自己创建自己的唯一实例。

3、单例类必须给所有其他对象提供这一实例。

```
// 单例模式的演示类
public class SingletonPatternDemo {
    public static void main(String[] args) {
        // 获取唯一可用的对象
        SingleObject object = SingleObject.getInstance();

        // 显示消息
        object.showMessage();
    }
}

// 单例类：SingleObject
public class SingleObject {
    // 创建 SingleObject 的一个静态实例
    private static SingleObject instance = new SingleObject();

    // 让构造函数为 private，这样该类就不会被实例化
    private SingleObject() {}

    // 获取唯一可用的对象
```

```
    public static SingleObject getInstance() {
        return instance;
    }

    // 显示消息的方法
    public void showMessage() {
        System.out.println("Hello World!");
    }
}
```

2、工厂模式

通过一个工厂类来实现对象的创建，而无需直接暴露对象的创建逻辑给客户端。

意图：定义一个创建对象的接口，让其子类自己决定实例化哪一个工厂类，工厂模式使其创建过程延迟到子类进行。

主要解决：主要解决接口选择的问题。

何时使用：我们明确地计划不同条件下创建不同实例时。

如何解决：让其子类实现工厂接口，返回的也是一个抽象的产品。

关键代码：创建过程在其子类执行。

优点： 1、一个调用者想创建一个对象，只要知道其名称就可以了。 2、扩展性高，如果想增加一个产品，只要扩展一个工厂类就可以。 3、屏蔽产品的具体实现，调用者只关心产品的接口。

```
// 主程序：演示工厂模式的使用
public class FactoryPatternDemo {
    public static void main(String[] args) {
        ShapeFactory shapeFactory = new ShapeFactory();

        // 获取 Circle 对象，并调用其 draw 方法
        Shape shape1 = shapeFactory.getShape("CIRCLE");
        shape1.draw();

        // 获取 Rectangle 对象，并调用其 draw 方法
        Shape shape2 = shapeFactory.getShape("RECTANGLE");
        shape2.draw();

        // 获取 Square 对象，并调用其 draw 方法
        Shape shape3 = shapeFactory.getShape("SQUARE");
        shape3.draw();
    }
}

// 接口：Shape（形状）
```

```java
public interface Shape {
    void draw(); // 抽象方法：绘制形状
}

// 实现类：Circle（圆形）
public class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Inside Circle::draw() method.");
    }
}

// 实现类：Rectangle（矩形）
public class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Inside Rectangle::draw() method.");
    }
}

// 实现类：Square（正方形）
public class Square implements Shape {
    @Override
    public void draw() {
        System.out.println("Inside Square::draw() method.");
    }
}

// 工厂类：ShapeFactory（形状工厂）
public class ShapeFactory {
    // 获取形状对象的方法
    public Shape getShape(String shapeType) {
        if (shapeType == null) {
            return null;
        }
        if (shapeType.equalsIgnoreCase("CIRCLE")) {
            return new Circle();
        } else if (shapeType.equalsIgnoreCase("RECTANGLE")) {
            return new Rectangle();
        } else if (shapeType.equalsIgnoreCase("SQUARE")) {
            return new Square();
        }
        return null;
    }
}
```

还有简单工厂模式、抽象工厂模式、多例模式、建造模式和原型模式。

结构型模式

描述如何将类或对象结合在一起形成更大的结构。继承的概念被用来组合接口和定义组合对象获得新功能的方式。

1、适配器模式

意图：将一个类的接口转换成客户希望的另外一个接口。适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

主要解决：主要解决在软件系统中，常常要将一些"现存的对象"放到新的环境中，而新环境要求的接口是现对象不能满足的。

何时使用： 1、系统需要使用现有的类，而此类的接口不符合系统的需要。 2、想要建立一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作，这些源类不一定有一致的接口。 3、通过接口转换，将一个类插入另一个类系中。

关键代码：适配器继承或依赖已有的对象，实现想要的目标接口。

```
// 主程序：演示适配器模式的功能
public class AdapterPatternDemo {
    public static void main(String[] args) {
        AudioPlayer audioPlayer = new AudioPlayer();

        // 播放各种音频格式的文件
        audioPlayer.play("mp3", "beyond the horizon.mp3");
        audioPlayer.play("mp4", "alone.mp4");
        audioPlayer.play("vlc", "far far away.vlc");
        audioPlayer.play("avi", "mind me.avi");
    }
}

// 媒体播放器接口：基础接口
public interface MediaPlayer {
    public void play(String audioType, String fileName);
}

// 高级媒体播放器接口：支持更多格式
public interface AdvancedMediaPlayer {
    public void playVlc(String fileName);
    public void playMp4(String fileName);
}

// 高级媒体播放器的具体实现：VLC播放器
public class VlcPlayer implements AdvancedMediaPlayer {
```

```
27      @Override
28      public void playVlc(String fileName) {
29          System.out.println("Playing vlc file. Name: " + fileName);
30      }
31
32      @Override
33      public void playMp4(String fileName) {
34          // 什么也不做
35      }
36  }
37
38  // 高级媒体播放器的具体实现：MP4播放器
39  public class Mp4Player implements AdvancedMediaPlayer {
40      @Override
41      public void playVlc(String fileName) {
42          // 什么也不做
43      }
44
45      @Override
46      public void playMp4(String fileName) {
47          System.out.println("Playing mp4 file. Name: " + fileName);
48      }
49  }
50
51  // 适配器类：将高级媒体播放器适配为基础媒体播放器
52  public class MediaAdapter implements MediaPlayer {
53      AdvancedMediaPlayer advancedMusicPlayer;
54
55      public MediaAdapter(String audioType) {
56          if (audioType.equalsIgnoreCase("vlc")) {
57              advancedMusicPlayer = new VlcPlayer();
58          } else if (audioType.equalsIgnoreCase("mp4")) {
59              advancedMusicPlayer = new Mp4Player();
60          }
61      }
62
63      @Override
64      public void play(String audioType, String fileName) {
65          if (audioType.equalsIgnoreCase("vlc")) {
66              advancedMusicPlayer.playVlc(fileName);
67          } else if (audioType.equalsIgnoreCase("mp4")) {
68              advancedMusicPlayer.playMp4(fileName);
69          }
70      }
71  }
72
73  // 音频播放器：基础媒体播放器实现类
```

```
74  public class AudioPlayer implements MediaPlayer {
75      MediaAdapter mediaAdapter;
76
77      @Override
78      public void play(String audioType, String fileName) {
79          // 播放 mp3 文件的内置支持
80          if (audioType.equalsIgnoreCase("mp3")) {
81              System.out.println("Playing mp3 file. Name: " + fileName);
82          }
83          // MediaAdapter 提供了播放其他文件格式的支持
84          else if (audioType.equalsIgnoreCase("vlc") ||
    audioType.equalsIgnoreCase("mp4")) {
85              mediaAdapter = new MediaAdapter(audioType);
86              mediaAdapter.play(audioType, fileName);
87          }
88          // 不支持的文件格式
89          else {
90              System.out.println("Invalid media. " + audioType + " format not
    supported");
91          }
92      }
93  }
```

2、桥接模式

意图：抽象部分与实现部分分离，使它们都可以独立的变化。

主要解决：在有多种可能会变化的情况下，用**继承会造成类爆炸**问题，扩展起来不灵活。

何时使用：实现系统可能有多个角度分类，每一种角度都可能变化。

如何解决：把这种多角度分类分离出来，让它们独立变化，减少它们之间耦合。这种减少耦合实际上符合了**开-闭原则。**

关键代码：抽象类依赖实现类。

```
1   // 桥接模式的演示类
2   public class BridgePatternDemo {
3       public static void main(String[] args) {
4           // 创建带有不同颜色的圆
5           Shape redCircle = new Circle(100, 100, 10, new RedCircle());
6           Shape greenCircle = new Circle(100, 100, 10, new GreenCircle());
7
8           // 绘制圆形
9           redCircle.draw();
10          greenCircle.draw();
11      }
```

```
}

// 抽象类：Shape（形状）
public abstract class Shape {
    protected DrawAPI drawAPI; // 桥接的接口

    protected Shape(DrawAPI drawAPI) {
        this.drawAPI = drawAPI;
    }

    public abstract void draw(); // 抽象的绘制方法
}

// 具体的形状类：Circle（圆形）
public class Circle extends Shape {
    private int x, y, radius;

    public Circle(int x, int y, int radius, DrawAPI drawAPI) {
        super(drawAPI);
        this.x = x;
        this.y = y;
        this.radius = radius;
    }

    @Override
    public void draw() {
        drawAPI.drawCircle(radius, x, y); // 使用 DrawAPI 绘制圆
    }
}

// 接口：DrawAPI（绘制接口）
public interface DrawAPI {
    void drawCircle(int radius, int x, int y); // 绘制圆的方法
}

// 具体实现类：GreenCircle（绿色圆形）
public class GreenCircle implements DrawAPI {
    @Override
    public void drawCircle(int radius, int x, int y) {
        System.out.println("Drawing Circle [ color: green, radius: " + radius
+ ", x: " + x + ", y: " + y + " ]");
    }
}

// 具体实现类：RedCircle（红色圆形）
public class RedCircle implements DrawAPI {
    @Override
```

```
    public void drawCircle(int radius, int x, int y) {
        System.out.println("Drawing Circle [ color: red, radius: " + radius +
", x: " + x + ", y: " + y + " ]");
    }
}
```

3、装饰器模式

意图：动态地给一个对象添加一些额外的职责。就增加功能来说，装饰器模式相比生成子类更为灵活。

主要解决：一般的，我们为了扩展一个类经常使用继承方式实现，由于继承为类引入静态特征，并且随着扩展功能的增多，子类会很膨胀。

何时使用：在不想增加很多子类的情况下扩展类。

如何解决：将具体功能职责划分，同时继承装饰者模式。

关键代码： 1、Component 类充当抽象角色，不应该具体实现。 2、修饰类引用和继承 Component 类，具体扩展类重写父类方法。

```
// 定义 Shape 接口，包含 draw 方法
public interface Shape {
    void draw();
}

// Circle 类，实现了 Shape 接口
public class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Shape: Circle");
    }
}

// Rectangle 类，实现了 Shape 接口
public class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Shape: Rectangle");
    }
}

// 抽象装饰器类，继承自 Shape 接口
public abstract class ShapeDecorator implements Shape {
    protected Shape decoratedShape; // 持有被装饰对象

    public ShapeDecorator(Shape decoratedShape) {
```

```
        this.decoratedShape = decoratedShape;
    }

    @Override
    public void draw() {
        // 调用被装饰对象的 draw 方法
        decoratedShape.draw();
    }
}

// RedShapeDecorator 类，继承 ShapeDecorator，增加红色边框功能
public class RedShapeDecorator extends ShapeDecorator {
    public RedShapeDecorator(Shape decoratedShape) {
        super(decoratedShape);
    }

    @Override
    public void draw() {
        // 调用基础功能
        decoratedShape.draw();
        // 添加红色边框的附加功能
        setRedBorder(decoratedShape);
    }

    private void setRedBorder(Shape decoratedShape) {
        System.out.println("Border Color: Red");
    }
}

// 测试装饰器模式的运行效果
public class DecoratorPatternDemo {
    public static void main(String[] args) {
        // 创建基础对象
        Shape circle = new Circle();
        Shape rectangle = new Rectangle();

        // 使用装饰器包装基础对象
        ShapeDecorator redCircle = new RedShapeDecorator(circle);
        ShapeDecorator redRectangle = new RedShapeDecorator(rectangle);

        // 调用原始对象的功能
        System.out.println("Circle with normal border");
        circle.draw();

        // 调用装饰器对象的功能（基础功能 + 红色边框）
        System.out.println("\nCircle of red border");
        redCircle.draw();
```

```
74
75          System.out.println("\nRectangle of red border");
76          redRectangle.draw();
77      }
78  }
```

4、代理模式

意图：为其他对象提供一种代理以控制对这个对象的访问。

主要解决：在直接访问对象时带来的问题，比如说：要访问的对象在远程的机器上。在面向对象系统中，有些对象由于某些原因（比如对象创建开销很大，或者某些操作需要安全控制，或者需要进程外的访问），直接访问会给使用者或者系统结构带来很多麻烦，我们可以在访问此对象时加上一个对此对象的访问层。

何时使用：想在访问一个类时做一些控制。

如何解决：增加中间层。

关键代码：实现与被代理类组合。

注意事项： 1、和适配器模式的区别：适配器模式主要改变所考虑对象的接口，而代理模式不能改变所代理类的接口。 2、和装饰器模式的区别：装饰器模式为了增强功能，而代理模式是为了加以控制。

```java
// 接口：Image（图像接口）
public interface Image {
    void display(); // 定义了一个展示图片的方法
}

// 真实类：RealImage（实际图像类）
public class RealImage implements Image {
    private String fileName;

    // 构造函数：加载图像
    public RealImage(String fileName) {
        this.fileName = fileName;
        loadFromDisk(fileName);
    }

    @Override
    public void display() {
        System.out.println("Displaying " + fileName);
    }

    // 模拟从磁盘加载图像的耗时操作
    private void loadFromDisk(String fileName) {
        System.out.println("Loading " + fileName);
```

```
    }
}

// 代理类：ProxyImage（代理图像类）
public class ProxyImage implements Image {
    private RealImage realImage;
    private String fileName;

    // 构造函数：初始化文件名
    public ProxyImage(String fileName) {
        this.fileName = fileName;
    }

    @Override
    public void display() {
        // 延迟加载真实图像对象
        if (realImage == null) {
            realImage = new RealImage(fileName);
        }
        realImage.display();
    }
}

// 演示类：ProxyPatternDemo（代理模式演示）
public class ProxyPatternDemo {
    public static void main(String[] args) {
        Image image = new ProxyImage("test_10mb.jpg");

        // 第一次调用会加载图像
        image.display();
        System.out.println("");

        // 第二次调用不需要加载图像
        image.display();
    }
}
```

5、外观模式

意图：为子系统中的一组接口提供一个一致的界面，外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

主要解决：降低访问复杂系统的内部子系统时的复杂度，简化客户端之间的接口。

何时使用：1、客户端不需要知道系统内部的复杂联系，整个系统只需提供一个"接待员"即可。2、定义系统的入口。

如何解决：客户端不与系统耦合，外观类与系统耦合。

关键代码：在客户端和复杂系统之间再加一层，这一层将调用顺序、依赖关系等处理好。

```
// 外观模式演示类
public class FacadePatternDemo {
    public static void main(String[] args) {
        // 创建 ShapeMaker 实例
        ShapeMaker shapeMaker = new ShapeMaker();

        // 调用 ShapeMaker 的方法绘制不同的形状
        shapeMaker.drawCircle();
        shapeMaker.drawRectangle();
        shapeMaker.drawSquare();
    }
}

// 接口：Shape（形状）
public interface Shape {
    void draw(); // 抽象方法：绘制形状
}

// 实现类：Circle（圆形）
public class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Circle::draw()");
    }
}

// 实现类：Rectangle（矩形）
public class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Rectangle::draw()");
    }
}

// 实现类：Square（正方形）
public class Square implements Shape {
    @Override
    public void draw() {
        System.out.println("Square::draw()");
    }
}

// 外观类：ShapeMaker（形状创建器）
```

```java
public class ShapeMaker {
    private Shape circle;
    private Shape rectangle;
    private Shape square;

    // 构造方法：初始化具体形状的对象
    public ShapeMaker() {
        circle = new Circle();
        rectangle = new Rectangle();
        square = new Square();
    }

    // 提供绘制 Circle 的方法
    public void drawCircle() {
        circle.draw();
    }

    // 提供绘制 Rectangle 的方法
    public void drawRectangle() {
        rectangle.draw();
    }

    // 提供绘制 Square 的方法
    public void drawSquare() {
        square.draw();
    }
}
```

特性	外观模式（Facade）	工厂模式（Facto
目的	简化子系统的接口，对外提供一个统一的接口。	简化对象的创建，
关注点	对系统操作的简化，聚合多个类或方法提供简单的调用方式。	对对象创建的简化
适用场景	子系统复杂，客户端只需要与外观类交互，而不需要深入子系统。	客户端需要创建对造方法或逻辑。
核心实现	聚合了子系统的多个类和方法，提供统一接口，隐藏内部复杂性。	提供一个工厂类，实例。
是否与子系统交互	外观模式直接与多个子系统交互，封装复杂的调用流程。	工厂模式不直接与的对象实例。
实例化的粒度	关注方法调用流程，不涉及创建实例。	关注对象的实例化

还有缺省适配器模式、合成模式、享元模式。

行为型模式

行为型模式是对在不同的对象之间划分责任和算法的抽象化。这些设计模式特别关注对象之间的通信。

1、观察者模式

意图：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

主要解决：一个对象状态改变给其他对象通知的问题，而且要考虑到易用和低耦合，保证高度的协作。

何时使用：一个对象（目标对象）的状态发生改变，所有的依赖对象（观察者对象）都将得到通知，进行广播通知。

关键代码：在抽象类里有一个 ArrayList 存放观察者们。

```
// 主程序入口
public class ObserverPatternDemo {
    public static void main(String[] args) {
        Subject subject = new Subject();

        // 添加观察者
        new HexaObserver(subject);
        new OctalObserver(subject);
        new BinaryObserver(subject);

        // 改变状态
        System.out.println("First state change: 15");
        subject.setState(15);

        System.out.println("Second state change: 10");
        subject.setState(10);
    }
}

// 被观察者类
import java.util.ArrayList;
import java.util.List;

public class Subject {
    private List<Observer> observers = new ArrayList<>(); // 存储所有观察者
    private int state; // 当前状态

    public int getState() {
        return state;
    }

```

```
32      public void setState(int state) {
33          this.state = state;
34          notifyAllObservers(); // 通知所有观察者
35      }
36
37      public void attach(Observer observer) {
38          observers.add(observer); // 添加观察者
39      }
40
41      public void notifyAllObservers() {
42          for (Observer observer : observers) {
43              observer.update(); // 调用每个观察者的更新方法
44          }
45      }
46  }
47
48  // 抽象观察者类
49  public abstract class Observer {
50      protected Subject subject; // 持有Subject的引用
51
52      public abstract void update(); // 抽象更新方法
53  }
54
55  // 二进制观察者
56  public class BinaryObserver extends Observer {
57      public BinaryObserver(Subject subject) {
58          this.subject = subject;
59          this.subject.attach(this); // 将自己注册到Subject中
60      }
61
62      @Override
63      public void update() {
64          System.out.println("Binary String: " +
    Integer.toBinaryString(subject.getState()));
65      }
66  }
67
68  // 十六进制观察者
69  public class HexaObserver extends Observer {
70      public HexaObserver(Subject subject) {
71          this.subject = subject;
72          this.subject.attach(this); // 将自己注册到Subject中
73      }
74
75      @Override
76      public void update() {
```

```
        System.out.println("Hex String: " +
Integer.toHexString(subject.getState()).toUpperCase());
    }
}

// 八进制观察者
public class OctalObserver extends Observer {
    public OctalObserver(Subject subject) {
        this.subject = subject;
        this.subject.attach(this); // 将自己注册到Subject中
    }

    @Override
    public void update() {
        System.out.println("Octal String: " +
Integer.toOctalString(subject.getState()));
    }
}
```

2、模板模式

意图：定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

主要解决：一些方法通用，却在每一个子类都重新写了这一方法。

何时使用：有一些通用的方法。

如何解决：将这些通用算法抽象出来。

关键代码：在抽象类实现，其他步骤在子类实现。

```
// 抽象类：Game（定义游戏的模板）
public abstract class Game {
    // 抽象方法，具体子类需实现
    abstract void initialize();
    abstract void startPlay();
    abstract void endPlay();

    // 模板方法，定义游戏的框架流程
    public final void play() {
        // 初始化游戏
        initialize();
        // 开始游戏
        startPlay();
        // 结束游戏
        endPlay();
```

```java
    }
}

// 具体实现类：Cricket（板球游戏）
public class Cricket extends Game {
    @Override
    void initialize() {
        System.out.println("Cricket Game Initialized! Start playing.");
    }

    @Override
    void startPlay() {
        System.out.println("Cricket Game Started. Enjoy the game!");
    }

    @Override
    void endPlay() {
        System.out.println("Cricket Game Finished!");
    }
}

// 具体实现类：Football（足球游戏）
public class Football extends Game {
    @Override
    void initialize() {
        System.out.println("Football Game Initialized! Start playing.");
    }

    @Override
    void startPlay() {
        System.out.println("Football Game Started. Enjoy the game!");
    }

    @Override
    void endPlay() {
        System.out.println("Football Game Finished!");
    }
}

// 演示类：TemplatePatternDemo（模板模式演示）
public class TemplatePatternDemo {
    public static void main(String[] args) {
        // 使用模板模式演示板球游戏
        Game game = new Cricket();
        game.play();
        System.out.println();

```

```
        // 使用模板模式演示足球游戏
        game = new Football();
        game.play();
    }
}
```

3、命令模式

意图：将一个请求封装成一个对象，从而使您可以用不同的请求对客户进行参数化。

主要解决：在软件系统中，行为请求者与行为实现者通常是一种紧耦合的关系，但某些场合，比如需要对行为进行记录、撤销或重做、事务等处理时，这种无法抵御变化的紧耦合的设计就不太合适。

何时使用：在某些场合，比如要对行为进行"记录、撤销/重做、事务"等处理，这种无法抵御变化的紧耦合是不合适的。在这种情况下，如何将"行为请求者"与"行为实现者"解耦？将一组行为抽象为对象，可以实现二者之间的松耦合。

如何解决：通过调用者调用接受者执行命令，顺序：调用者→命令→接受者。

关键代码：定义三个角色：1、received 真正的命令执行对象 2、Command 3、invoker 使用命令对象的入口

优点： 1、降低了系统耦合度。 2、新的命令可以很容易添加到系统中去。

```
import java.util.ArrayList;
import java.util.List;

// Command 接口
public interface Order {
    void execute();
}

// 请求类
public class Stock {
    private String name = "ABC";
    private int quantity = 10;

    public void buy() {
        System.out.println("Stock [ Name: " + name + ", Quantity: " +
quantity + " ] bought");
    }

    public void sell() {
        System.out.println("Stock [ Name: " + name + ", Quantity: " +
quantity + " ] sold");
    }
}
```

```java

// BuyStock 命令类
public class BuyStock implements Order {
    private Stock abcStock;

    public BuyStock(Stock abcStock) {
        this.abcStock = abcStock;
    }

    @Override
    public void execute() {
        abcStock.buy();
    }
}

// SellStock 命令类
public class SellStock implements Order {
    private Stock abcStock;

    public SellStock(Stock abcStock) {
        this.abcStock = abcStock;
    }

    @Override
    public void execute() {
        abcStock.sell();
    }
}

// 调用者类
public class Broker {
    private List<Order> orderList = new ArrayList<>();

    public void takeOrder(Order order) {
        orderList.add(order);
    }

    public void placeOrders() {
        for (Order order : orderList) {
            order.execute();
        }
        orderList.clear();
    }
}

// 客户端代码
public class CommandPatternDemo {
```

```
    public static void main(String[] args) {
        Stock abcStock = new Stock();

        BuyStock buyStockOrder = new BuyStock(abcStock);
        SellStock sellStockOrder = new SellStock(abcStock);

        Broker broker = new Broker();
        broker.takeOrder(buyStockOrder);
        broker.takeOrder(sellStockOrder);

        broker.placeOrders();
    }
}
```

4、迭代器模式

意图：提供一种方法顺序访问一个聚合对象中各个元素, 而又无须暴露该对象的内部表示。

主要解决：不同的方式来遍历整个整合对象。

何时使用：遍历一个聚合对象。

如何解决：把在元素之间游走的责任交给迭代器，而不是聚合对象。

关键代码：定义接口：hasNext, next。

```
// Iterator 接口
public interface Iterator {
    public boolean hasNext();
    public Object next();
}

// Container 接口
public interface Container {
    public Iterator getIterator();
}

// NameRepository 类实现 Container 接口
public class NameRepository implements Container {
    public String[] names = {"Robert", "John", "Julie", "Lora"};

    @Override
    public Iterator getIterator() {
        return new NameIterator();
    }

    // 内部类实现 Iterator 接口
```

```
22      private class NameIterator implements Iterator {
23          int index;
24
25          @Override
26          public boolean hasNext() {
27              return index < names.length;
28          }
29
30          @Override
31          public Object next() {
32              if (this.hasNext()) {
33                  return names[index++];
34              }
35              return null;
36          }
37      }
38  }
39
40  // 客户端代码
41  public class IteratorPatternDemo {
42      public static void main(String[] args) {
43          NameRepository namesRepository = new NameRepository();
44
45          for (Iterator iter = namesRepository.getIterator(); iter.hasNext();) {
46              String name = (String) iter.next();
47              System.out.println("Name : " + name);
48          }
49      }
50  }
```

5、策略模式

意图：定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换。

主要解决：在有多种算法相似的情况下，使用 if...else 所带来的复杂和难以维护。

何时使用：一个系统有许多许多类，而区分它们的只是他们直接的行为。

如何解决：将这些算法封装成一个一个的类，任意地替换。

关键代码：实现同一个接口。

优点： 1、算法可以自由切换。2、避免使用多重条件判断。 3、扩展性良好。

```
1   // 策略模式示例：动态更换算法以执行操作
2   // 定义策略接口
3   interface MathAlgorithm {
4       // 策略接口定义计算方法
```

```
    int calculate(int num1, int num2);
}

// 策略1：加法实现
class MathAdd implements MathAlgorithm {
    @Override
    public int calculate(int num1, int num2) {
        return num1 + num2; // 返回加法结果
    }
}

// 策略2：减法实现
class MathSubtract implements MathAlgorithm {
    @Override
    public int calculate(int num1, int num2) {
        return num1 - num2; // 返回减法结果
    }
}

// 策略3：乘法实现
class MathMultiply implements MathAlgorithm {
    @Override
    public int calculate(int num1, int num2) {
        return num1 * num2; // 返回乘法结果
    }
}

// 环境类：使用策略模式执行算法
class MathContext {
    private MathAlgorithm algorithm; // 当前算法策略

    // 构造函数，注入具体的策略
    public MathContext(MathAlgorithm strategy) {
        this.algorithm = strategy;
    }

    // 执行计算
    public int execute(int num1, int num2) {
        return algorithm.calculate(num1, num2); // 调用策略方法
    }
}

// 测试类：演示策略模式
public class Main {
    public static void main(String[] args) {
        // 使用加法策略
        MathContext context = new MathContext(new MathAdd());
```

```
        System.out.println("10 + 5 = " + context.execute(10, 5));

        // 切换到减法策略
        context = new MathContext(new MathSubtract());
        System.out.println("10 - 5 = " + context.execute(10, 5));

        // 切换到乘法策略
        context = new MathContext(new MathMultiply());
        System.out.println("10 * 5 = " + context.execute(10, 5));
    }
}
```

6、空对象模式

在空对象模式中，我们创建一个指定各种要执行的操作的抽象类和扩展该类的实体类，还创建一个未对该类做任何实现的空对象类，该空对象类将无缝地使用在需要检查空值的地方。

```
// AbstractEmployee.java
abstract class AbstractEmployee {
    protected String name;
    public abstract boolean isNull();
    public abstract String getName();
}

// Programmer.java
class Programmer extends AbstractEmployee {
    public Programmer(String name) {
        this.name = name;
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public boolean isNull() {
        return false;
    }
}

// NullCustomer.java
class NullCustomer extends AbstractEmployee {
    @Override
    public String getName() {
```

```
29            return "Not Available";
30        }
31
32        @Override
33        public boolean isNull() {
34            return true;
35        }
36    }
37
38    // EmployeeFactory.java
39    class EmployeeFactory {
40        public static final String[] names = { "Rob", "Joe", "Jack" };
41
42        public static AbstractEmployee getCustomer(String name) {
43            for (int i = 0; i < names.length; i++) {
44                if (names[i].equalsIgnoreCase(name)) {
45                    return new Programmer(name);
46                }
47            }
48            return new NullCustomer();
49        }
50    }
51
52    // Main.java
53    public class Main {
54        public static void main(String[] args) {
55            AbstractEmployee emp = EmployeeFactory.getCustomer("Rob");
56            AbstractEmployee emp2 = EmployeeFactory.getCustomer("Bob");
57            AbstractEmployee emp3 = EmployeeFactory.getCustomer("Jack");
58            AbstractEmployee emp4 = EmployeeFactory.getCustomer("Tom");
59
60            System.out.println(emp.getName());
61            System.out.println(emp2.getName());
62            System.out.println(emp3.getName());
63            System.out.println(emp4.getName());
64        }
65    }
```

7、MVC模式

Model（模型） - 模型代表一个存取数据的对象。它也可以带有逻辑，在数据变化时更新控制器。

View（视图） - 视图代表模型包含的数据的可视化。

Controller（控制器） - 控制器作用于模型和视图上。它控制数据流向模型对象，并在数据变化时更新视图。它使视图与模型分离开。

```java
// Model-View-Controller (MVC) 设计模式实现

// Student.java - 模型 (Model)
public class Student {
    private String rollNo; // 学号
    private String name;   // 姓名

    // 获取学号
    public String getRollNo() {
        return rollNo;
    }

    // 设置学号
    public void setRollNo(String rollNo) {
        this.rollNo = rollNo;
    }

    // 获取姓名
    public String getName() {
        return name;
    }

    // 设置姓名
    public void setName(String name) {
        this.name = name;
    }
}

// StudentView.java - 视图 (View)
public class StudentView {
    // 打印学生详细信息
    public void printStudentDetails(String studentName, String studentRollNo)
{
        System.out.println("学生信息: ");
        System.out.println("姓名: " + studentName);
        System.out.println("学号: " + studentRollNo);
    }
}

// StudentController.java - 控制器 (Controller)
public class StudentController {
    private Student model; // 模型
    private StudentView view; // 视图

    // 构造函数，初始化模型和视图
    public StudentController(Student model, StudentView view) {
        this.model = model;
```

```
        this.view = view;
    }

    // 设置学生姓名
    public void setStudentName(String name) {
        model.setName(name);
    }

    // 获取学生姓名
    public String getStudentName() {
        return model.getName();
    }

    // 设置学生学号
    public void setStudentRollNo(String rollNo) {
        model.setRollNo(rollNo);
    }

    // 获取学生学号
    public String getStudentRollNo() {
        return model.getRollNo();
    }

    // 更新视图，显示模型中的数据
    public void updateView() {
        view.printStudentDetails(model.getName(), model.getRollNo());
    }
}

// MVCPatternDemo.java - 主程序
public class MVCPatternDemo {
    public static void main(String[] args) {
        // 从模拟数据库中获取学生记录
        Student model = retrieveStudentFromDatabase();

        // 创建一个视图，用于显示学生信息
        StudentView view = new StudentView();

        // 创建控制器，并传入模型和视图
        StudentController controller = new StudentController(model, view);

        // 显示初始学生信息
        controller.updateView();

        // 更新模型中的数据
        controller.setStudentName("John");

```

```
94            // 将更新后的数据反映到视图中
95            controller.updateView();
96        }
97
98        // 模拟从数据库中获取学生记录
99        private static Student retrieveStudentFromDatabase() {
100           Student student = new Student();
101           student.setName("Robert");
102           student.setRollNo("10");
103           return student;
104       }
105   }
```

还有不变模式、责任链模式、备忘录模式、状态模式、访问者模式、解释器模式和调停者模式。

敏捷开发

测试驱动

- 测试驱动开发是一种先写测试，再写代码的开发方法，强调通过测试来驱动设计和实现。
- Prep code -> test code -> real code

软件设计的一般原则

- **开-闭原则**
  - 软件实体（类、模块、函数等）应该对扩展开放，对修改关闭。
  - 不允许修改系统的抽象层，允许拓展系统的实现层
- **里氏代换原则**
  - 如果使用一个父类，那么一定可以用子类替换，对系统的功能没有影响
  - 尽量从抽象类继承，不从具体类继承
- **依赖倒置原则**
  - 高层模块不应该依赖于低层模块，两者都应该依赖于抽象接口；抽象接口不应该依赖于具体实现，具体实现应该依赖于抽象接口。
  - 要求客户端依赖于抽象耦合

```
1   private List list = new ArrayList();
2   而不是
3   private ArrayList list = new ArrayList();
```

- **接口隔离原则**
  - 建立最小的接口
  - 不应该强迫一个类实现它不需要的接口，应该将接口拆分成更小和更具体的部分，以便客户端只需要知道它们感兴趣的部分。
  - 调用者只能访问自己的方法，不能访问不该访问的方法
- **合成/聚合原则**
  - 尽可能少继承，多复用
  - Has-A不继承，is-A继承；永远不会出现子类需要替换成另一个类的子类的情况；子类有拓展超类的责任
- **迪米特法则**
  - 一个对象应该对其他对象有尽可能少的了解，通常称为“最少知识原则”。
- **单一职责原则**
  - 一个类只应该有一个引起它变化的原因。

```
1   void change(String username, String address) x
2
3   void change(String username)
4   void change(String address) √
```

【例】埃拉托斯特尼筛法：素数产生程序

Web