Summary Notes

Bubble Sort (冒泡排序)

- **最好情况:** $O(n)$ （已经排序的数组）
- **平均情况:** $O(n^2)$
- **最坏情况:** $O(n^2)$
- **空间复杂度:** $O(1)$ （原地排序）
- **稳定性:** 稳定

Insertion Sort (插入排序)

- **最好情况:** $O(n)$ （已经排序的数组）
- **平均情况:** $O(n^2)$
- **最坏情况:** $O(n^2)$
- **空间复杂度:** $O(1)$ （原地排序）
- **稳定性:** 稳定

Merge Sort (归并排序)

- **最好情况:** $O(n \log n)$
- **平均情况:** $O(n \log n)$
- **最坏情况:** $O(n \log n)$
- **空间复杂度:** $O(n)$ （需要额外的数组存储合并的结果）
- **稳定性:** 稳定

Heap Sort (堆排序)

- **最好情况:** $O(n \log n)$
- **平均情况:** $O(n \log n)$
- **最坏情况:** $O(n \log n)$
- **空间复杂度:** $O(1)$ （原地排序）
- **稳定性:** 不稳定

Quick Sort (快速排序)

- **最好情况:** $O(n \log n)$
- **平均情况:** $O(n \log n)$
- **最坏情况:** $O(n^2)$ （最坏情况是选择了一个极端不平衡的划分，比如已经排序的数组）
- **空间复杂度:** $O(\log n)$ （递归栈空间）
- **稳定性:** 不稳定

Selection Sort (选择排序)

- **最好情况:** $O(n^2)$
- **平均情况:** $O(n^2)$
- **最坏情况:** $O(n^2)$
- **空间复杂度:** $O(1)$ （原地排序）
- **稳定性:** 不稳定

Heap Sort 是一种基于堆数据结构的排序算法，其时间复杂度在不同情况下如下：

1. **最坏情况 (Worst Case)**：O(n log n)
2. **平均情况 (Average Case)**：O(n log n)
3. **最好情况 (Best Case)**：O(n log n)

让我们详细分析这些复杂度：

最坏情况 (Worst Case)

在最坏情况下，Heap Sort 的时间复杂度为 O(n log n)。在堆排序中，构建堆需要 O(n) 的时间，而每次从堆中提取最大元素并重新调整堆的操作需要 O(log n) 的时间。因为我们需要进行 n 次提取和调整操作，所以总的时间复杂度为 O(n log n)。

平均情况 (Average Case)

平均情况下，Heap Sort 的时间复杂度同样为 O(n log n)。这是因为构建堆和调整堆的过程在平均情况下也需要相同数量的操作，即 O(n log n)。

最好情况 (Best Case)

在最好情况下，Heap Sort 的时间复杂度仍然是 O(n log n)。即使输入数据已经有序，Heap Sort 仍然需要构建堆和进行 n 次提取和调整操作，因此其时间复杂度没有变化。

继承 (Inheritance)

继承是面向对象编程中一种机制，允许一个类（子类）从另一个类（父类）获得属性和方法。通过继承，子类可以重用父类的代码，并可以添加新的属性和方法或重写父类的方法，从而增强代码的可重用性和可维护性。

Inheritance is a mechanism in object-oriented programming that allows one class (the subclass) to inherit attributes and methods from another class (the superclass). Through inheritance, the subclass can reuse the code of the superclass and add new attributes and methods or override existing ones, thus enhancing code **reusability** and **maintainability**.

封装 (Encapsulation)

封装是将对象的属性和方法隐藏起来，仅对外暴露必要的接口。通过封装，可以保护对象的内部状态，防止外部代码直接访问和修改，从而提高代码的安全性和灵活性。

Encapsulation is the practice of hiding an object's attributes and methods, exposing only the necessary interfaces to the outside world. Encapsulation protects the internal state of the object, preventing direct access and modification by external code, thereby increasing code security and flexibility.

多态 (Polymorphism)

多态是指在面向对象编程中，不同类的对象可以通过相同的接口调用各自的方法，从而实现同一操作在不同对象中的不同表现。多态提高了代码的灵活性和可扩展性。

Polymorphism refers to the ability in object-oriented programming for different classes to be treated through the same interface, allowing the same operation to behave differently on different objects. Polymorphism enhances code flexibility and extensibility.

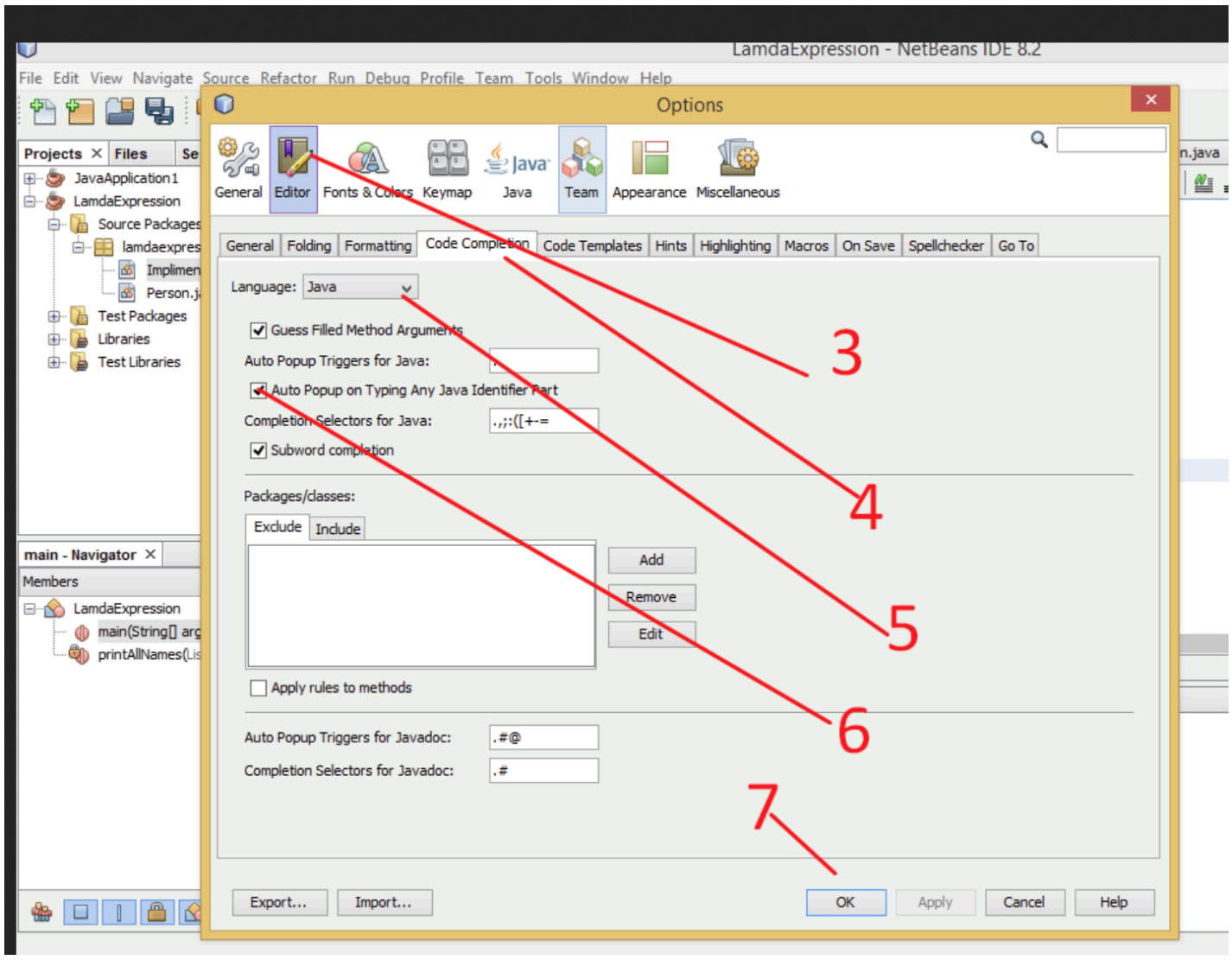
抽象 (Abstraction)

抽象是指在面向对象编程中，通过抽取对象的共有特征（属性和方法），定义出一个通用的类（抽象类），以简化复杂系统。抽象类不能实例化，只能作为其他类的基类。

Abstraction in object-oriented programming is the process of defining a general class (abstract class) by extracting common characteristics (attributes and methods) from multiple objects to simplify complex systems. An abstract class cannot be instantiated and is meant to serve as a base class for other classes.

这四个概念是面向对象编程的核心，帮助开发人员创建模块化、可维护和可扩展的代码。

Netbean 自动补全 tool-options-editor-code complement-java



F6 //运行项目

psvm + Tab //生成Main方法

sout + Tab //生成输出语句

Alt + Enter //当系统报错时，按下此组合可以查看系统提示

Ctrl + \ //任何地方按下此组合键，均会提示相应的参考字段

Ctrl + E //删除当前行

Ctrl + R //变量重命名

Ctrl + O //快速打开类

Ctrl + W //快速关闭当前窗口

Ctrl + K //组合键自动打出字符串，每按一次打出一个新串，串序自下向上

Ctrl + Shift + I //导入所需包

Alt + Shift + F //格式化代码

Ctrl + / //注释/取消注,此功能支持多行注释，但首先需选中所要注释行

Ctrl + Enter //增加空白行，光标不移动

Alt + Insert //插入代码（包括构造函数，setter和getter方法等）

Alt + Shift + O //转到类、文件

Ctrl + Shift + UP/DOWN //复制当前行到下一行，光标不动

Ctrl + Delete //删除光标后一个字符串

Ctrl + Backspace //删除光标前一个字符串

Ctrl + Shift + Right　　//向右逐个选中
Ctrl + Shift + Left　　//向左逐个选中

Ctrl + (小键盘)-　　//折叠（隐藏）代码块
Ctrl + (小键盘)+　　//展开已折叠的代码块

Shift + Esc　　//最大化窗口（切换）

Ctrl + Shift + J　　//插入国际化字符串

编译、测试和运行

F9 编译选定的包或文件
F11 生成主项目
Shift + F11 清理并生成主项目
Ctrl + Q 设置请求参数
Ctrl + Shift-U 创建 JUnit 测试
Ctrl + F6/Alt-F6 为文件/项目运行JUnit测试
Shift + F6/F6 运行主项目/文件

调试

F5 开始调试主项目
F4 运行到文件中的光标位置
F7/F8 步入/越过
Ctrl + Shift + F5 开始调试当前文件
Ctrl + Shift + F6 开始为文件调试测试 (JU
Shift + F5/Ctrl + F5 停止/继续调试会话

Ctrl + F7 步出
Ctrl + Alt + 向上方向键 转至被调用的方法
Ctrl + Alt + 向下方向键 转至调用方法
Ctrl + F9 计算表达式的值
Ctrl + F8 切换断点
Ctrl + Shift + F8 新建断点
Ctrl + Shift + F7 新建监视
Ctrl + Shift + 5 显示 HTTP 监视器
Ctrl + Shift + 0 显示“搜索结果”窗口
Alt + Shift + 1 显示“局部变量”窗口
Alt + Shift + 2 显示“监视”窗口
Alt + Shift + 3 显示“调用栈”窗口
Alt + Shift + 4 显示“类”窗口
Alt + Shift + 5 显示“断点”窗口
Alt + Shift + 6 显示“会话”窗口
Ctrl + Shift + 6 切换到“执行”窗口
Alt + Shift + 7 切换到“线程”窗口
Alt + Shift + 8 切换到“源”窗口

Lecture3 - Abstract

1. An abstract method can only be contained in an abstract class.

抽象方法必须在抽象类中存在，也就是说如果一个类中存在`public abstract void myMethod();`

那么这个类必须被声明为抽象类`public abstract class MyAbstractClass { ... }`。

换句话说，如果一个类中包含了一个或多个抽象方法，那么这个类也必须是抽象类。这是因为抽象方法没有实现，普通类（非抽象类）不能包含未实现的方法。

2. subclasses of abstract classes

In an abstract subclass extended from an abstract super-class, we can choose:
a. to implement the inherited abstract methods OR
b. to postpone the constraint to implement the
abstract methods to its nonabstract subclasses.

在 Java 中，如果一个抽象子类继承自一个抽象父类，那么这个抽象子类可以选择：

a. 实现继承的抽象方法，或者
b. 将实现抽象方法的责任推迟到其非抽象的子类。

a. 实现继承的抽象方法

一个抽象子类可以**选择实现其继承自抽象父类的抽象方法**。这意味着该子类提供了这些方法的具体实现，使得这些方法在子类中可以被调用。例如：

```
abstract class Animal {
    public abstract void makeSound();
    }
```

```
abstract class Mammal extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Some generic mammal sound");
    }
}
```

b. 将实现抽象方法的责任推迟到其非抽象的子类

一个抽象子类也可以选择不实现继承的抽象方法，而是将实现的责任推迟到其具体的子类。这意味着抽象子类本身仍然是抽象的，并且它的子类必须实现这些抽象方法。例如：

```
abstract class Animal {
    public abstract void makeSound();
}

abstract class Mammal extends Animal {
    // 不实现 makeSound() 方法
}

class Dog extends Mammal {
    @Override
    public void makeSound() {
        System.out.println("Bark");
    }
}

class Cat extends Mammal {
    @Override
    public void makeSound() {
        System.out.println("Meow");
    }
}
```

3. A subclass can be abstract even if its superclass is concrete.

即使父类是具体的子类也可以是抽象的
For example, the Object class is concrete, but a subclass, GeometricObject, is abstract

A subclass can override a method from its concrete superclass to define it abstract

1. useful when we want to **force its subclasses to implement that method**, or
2. the implementation of the method in the superclass is **invalid** in the subclass

在 Java 中，一个子类可以**重写（**override）**其具体超类（非抽象类）中的方法并将其定义为抽象方法**。这样做有两个主要原因：

1. 强制子类实现该方法

当我们希望所有的子类都必须实现一个特定的方法时，可以在一个抽象子类中将这个方法定义为抽象方法。这样做的目的是确保任何继承这个抽象子类的具体子类都必须提供该方法的实现。

2. 超类中的方法在子类中无效

如果超类中的某个方法在子类的上下文中不适用或无效，可以通过将该方法定义为抽象方法来表示子类不应该使用超类的实现。这样，具体子类将被迫提供适合它们的实现。

```
public class Vehicle {
    public void startEngine() {
        System.out.println("Starting engine...");
    }
}

public abstract class ElectricVehicle extends Vehicle {
    @Override
    public abstract void startEngine();
}

public class Tesla extends ElectricVehicle {
    @Override
    public void startEngine() {
        System.out.println("Powering up electric motor...");
    }
}

public class NissanLeaf extends ElectricVehicle {
    @Override
    public void startEngine() {
        System.out.println("Turning on electric systems...");
    }
}
```

4. It is possible to define an abstract class that contains no abstract methods.

This class is used as a base class for defining new subclasses.

在 Java 中定义一个没有抽象方法的抽象类是完全可能的，并且这种类通常用作基础类来定义新的子类。这样的抽象类可以包含具体的方法、构造函数和字段，但不能直接实例化。

1. 为什么定义没有抽象方法的抽象类

a. 提供通用功能

这些抽象类通常用于**提供一些通用的功能和状态，供子类共享和使用**。这样，子类可以继承这些功能，而不必在每个子类中重复实现。

b. 防止直接实例化

通过将类定义为抽象类，可以**防止该类被直接实例化**。如果某个类本身并不代表一个具体的实现，而只是用作其他类的基础，这种方式是非常有用的。

5. An object cannot be created from abstract class:

An abstract class cannot be instantiated using the new operator:

`GeometricObject o =` `new GeometricObject();`

抽象类和接口一样都不能被实例化 只是用来定义一些common function and attribute，但是抽象类有构造器 目的是为了在具体实现子类实例化时通过**constructor chain**来正确的初始化子类的所有属性/字段

6. An abstract class can be used as a data type

抽象类可以作为 声明变量也就是数据类型 declare variable / data type

`GeometricObject c = new Circle(2);`

We can create an array whose elements are of `GeometricObject` type

```
GeometricObject[] geo = new GeometricObject[10];
// 在geo中的元素没有被实例化时 里面都是null
geo[0] = new Circle();
geo[1] = new Rectangle();
```

7. The abstract `Calendar` class and its `GregorianCalendar` subclass

An instance of `java.util.Date` represents a specific instant in time with millisecond precision

`Date` 类在 Java 中用于表示特定的时间点，精度为毫秒。它主要用于获取当前时间或指定的时间点。

```
Date currentDate = new Date();
System.out.println("Current Date: " + currentDate);
```

```
long timestamp = 1622547800000L;
Date specificDate = new Date(timestamp);
System.out.println("Specific Date: " + specificDate);
```

a. `java.util.Calendar` **is an abstract base class** for **extracting detailed information** such as year, month, date, hour, minute and second from a Date object for a specific calendar

`Calendar` 类是一个抽象基类，用于从 `Date` 对象中提取详细的信息，比如年、月、日、小时、分钟和秒。它允许进行复杂的日期操作，如日期计算和调整。

```
Calendar calendar = Calendar.getInstance();

calendar.setTime(currentDate);
int year = calendar.get(Calendar.YEAR);
int month = calendar.get(Calendar.MONTH) + 1; // 注意：月份从0开始
int day = calendar.get(Calendar.DAY_OF_MONTH);
System.out.println("Year: " + year + ", Month: " + month + ", Day: " + day);

calendar.add(Calendar.DAY_OF_MONTH, 5); // 增加5天
Date newDate = calendar.getTime();
System.out.println("New Date: " + newDate);
```

b. Subclasses of Calendar can implement specific calendar systems such as Gregorian calendar, Lunar Calendar and Jewish calendar.

`GregorianCalendar` 是 `Calendar` 类的具体子类，专门用于处理现代公历（格里高利历）。它是 Java 中最常用的日期类之一。

c.

`java.util.GregorianCalendar` is for the modern Gregorian calendar

```
// 默认构造函数:
GregorianCalendar gregorianCalendar = new GregorianCalendar();
Date date = gregorianCalendar.getTime();
System.out.println("Current Date from GregorianCalendar: " + date);

// 指定年、月、日的构造函数:
GregorianCalendar specificDate = new GregorianCalendar(2024, Calendar.JUNE, 10);
Date date = specificDate.getTime();
System.out.println("Specific Date from GregorianCalendar: " + date);
```

```
import java.util.*;
 public class TestCalendar {
 public static void main(String[] args) {
 // Construct a Gregorian calendar for the current date and time
 Calendar calendar = new GregorianCalendar();
 System.out.println("Current time is " + new Date());
 System.out.println("YEAR:\t" + calendar.get(Calendar.YEAR));
 System.out.println("MONTH:\t" + calendar.get(Calendar.MONTH));
 System.out.println("DATE:\t" + calendar.get(Calendar.DATE));
 System.out.println("HOUR:\t" + calendar.get(Calendar.HOUR));
 System.out.println("HOUR_OF_DAY:\t" + calendar.get(Calendar.HOUR_OF_DAY));
 System.out.println("MINUTE:\t" + calendar.get(Calendar.MINUTE));
 System.out.println("SECOND:\t" + calendar.get(Calendar.SECOND));
 System.out.println("DAY_OF_WEEK:\t" + calendar.get(Calendar.DAY_OF_WEEK));
 System.out.println("DAY_OF_MONTH:\t" + calendar.get(Calendar.DAY_OF_MONTH));
 System.out.println("DAY_OF_YEAR: " + calendar.get(Calendar.DAY_OF_YEAR));
 System.out.println("WEEK_OF_MONTH: " + calendar.get(Calendar.WEEK_OF_MONTH));
 System.out.println("WEEK_OF_YEAR: " + calendar.get(Calendar.WEEK_OF_YEAR));
 System.out.println("AM_PM: " + calendar.get(Calendar.AM_PM));
 // Construct a calendar for January 1, 2020
 Calendar calendar1 = new GregorianCalendar(2020, 0, 1);
 System.out.println("January 1, 2020 is a " +
dayNameOfWeek(calendar1.get(Calendar.DAY_OF_WEEK)) );
 }
public static String dayNameOfWeek(int dayOfWeek) {
 switch (dayOfWeek) {
 case 1: return "Sunday"; case 2: return "Monday"; case 3: return "Tuesday";
... case 7: return "Saturday";
 default: return null;
 }}}
```

月份参数的注意事项

在 `GregorianCalendar` 中，月份参数是从0开始的，这意味着：

- 0 对应 1 月
- 1 对应 2 月
- 2 对应 3 月
- ...
- 11 对应 12 月

```
// 输出日历
public static void main(String[] args) {
```

```
        Scanner scanner = new Scanner(System.in);

        // 获取年份和月份的输入
        System.out.print("input year: ");
        int year = scanner.nextInt();
        System.out.print("input month: ");
        int month = scanner.nextInt();

        // 创建GregorianCalendar对象
        Calendar calendar = new GregorianCalendar(year, month - 1, 1);

        // 获取该月的天数
        int daysInMonth = calendar.getActualMaximum(Calendar.DAY_OF_MONTH);

        // 打印月和年份
        System.out.println("\n" + year + " year " + (month) + " month");

        // 打印星期标题
        System.out.println("Sun Mon Tue Wed Thu Fri Sat");

        // 获取该月的第一天是星期几
        int firstDayOfWeek = calendar.get(Calendar.DAY_OF_WEEK);

        // 在第一个星期前面填充空格
        for (int i = 1; i < firstDayOfWeek; i++) {
            System.out.print("    ");
        }

        // 打印日期
        for (int day = 1; day <= daysInMonth; day++) {
            System.out.printf("%3d ", day);
            if ((day + firstDayOfWeek - 1) % 7 == 0) {
                System.out.println();
            }
        }

        // 换行
        System.out.println();
    }
```

```
GregorianCalendar specificDate = new GregorianCalendar(2024, 0, 10); // 2024年1月10日
Date date = specificDate.getTime();
System.out.println("January 10, 2024: " + date);
```

8. Interface

An interface is a class-like construct that contains **only abstract methods** and **constants**.

An interface is similar to an abstract class, but the intent of an interface is to specify behavior for objects.

- **行为规范**：定义对象可以执行的操作，强制实现这些操作。

  Behavior specification: Defines the operations an object can perform and enforces the implementation of these operations.

- **多态性**：允许通过接口引用不同实现类的对象，实现多态行为。

Polymorphism: Allows different implementation classes to be referenced through the interface, achieving polymorphic behavior.

- **解耦**：通过接口实现模块之间的低耦合，提高代码的灵活性和可维护性。

  Decoupling: Achieves low coupling between modules through interfaces, enhancing code flexibility and maintainability.

- **多重继承**：允许类实现多个接口，从而继承多个接口的行为规范。

  Multiple inheritance: Allows a class to implement multiple interfaces, thereby inheriting the behavior specifications of multiple interfaces.

接口与抽象类的比较

尽管接口与抽象类都可以用来定义一组抽象行为，但它们的用途和使用方式有所不同：

Although both interfaces and abstract classes can be used to define a set of abstract behaviors, their purposes and usage differ:

1. **接口**：
   - 接口主要用于定义行为规范，指定对象可以做什么。

     Interfaces are mainly used to define behavior specifications, specifying what an object can do.
   - 一个类可以实现多个接口，从而支持多重继承接口的方法。

     A class can implement multiple interfaces, thus supporting multiple inheritance of interface methods.

1. **抽象类**：
   - 抽象类用于表示类的通用概念，可以包含抽象方法和具体方法。

     Abstract classes are used to represent general concepts of a class and can contain both abstract and concrete methods.

- 一个类只能继承一个抽象类。

  A class can only inherit one abstract class.

```
// 定义一个接口，用于比较对象
public interface Comparable<T> {
    int compareTo(T o);
}

// 定义一个接口，用于表示可食用的对象
public interface Edible {
        public abstract String howToEat();
    void eat();
}

// 定义一个类，实现了两个接口
public class Apple implements Comparable<Apple>, Edible {
    private String variety;

    public Apple(String variety) {
        this.variety = variety;
    }

    @Override
    public int compareTo(Apple other) {
        return this.variety.compareTo(other.variety);
    }

    @Override
    public void eat() {
        System.out.println("Eating the apple of variety: " + variety);
```

```
    }

    @Override
    public String howToEat() {
            return "Chicken: Fry it";
        }

    public static void main(String[] args) {
        Apple apple1 = new Apple("Fuji");
        Apple apple2 = new Apple("Gala");

        System.out.println("Comparison result: " + apple1.compareTo(apple2));
        apple1.eat();
    }
}
```

9. Omitting Modifiers in Interfaces

All data fields are `public static final`

All methods are `public abstract`

A constant defined in an interface can be accessed using `InterfaceName.CONSTANT_NAME`, for example: `T1.K`

10. Interface is treated like a special class in Java

Like an abstract class, you **cannot create an instance** from an interface using the `new` operator

接口在 Java 中被视为一种特殊的类，具有以下特点：

Interfaces in Java are treated as special classes with the following characteristics:

- **独立的字节码文件**：每个接口会编译成一个独立的 `.class` 文件。

  Separate bytecode files: Each interface is compiled into a separate `.class` file.

- **不能直接实例化**：不能使用 `new` 操作符创建接口实例。

  Cannot be directly instantiated: You cannot create an instance of an interface using the `new` operator.

- **作为数据类型和类型转换的目标**：接口可以用作变量的数据类型和类型转换的目标。

  Used as data types and casting targets: Interfaces can be used as the data type for variables and as casting targets.

11. Comparable Interface

`Comparable` 接口定义在 `java.lang` 包中，用于提供对象的自然顺序。许多 Java 库中的类（例如 `String` 和 `Date`）都实现了 `Comparable` 接口，以便定义对象的自然排序顺序。

```
public class ComparableRectangle extends Rectangle implements Comparable {
 /** Construct a ComparableRectangle with specified properties */
 public ComparableRectangle(double width, double height) {
```

```
    super(width, height);
}
/** Implement the compareTo method defined in Comparable */
public int compareTo(Object o) {
    if (getArea() > ((ComparableRectangle)o).getArea())
        return 1;
    else if (getArea() < ((ComparableRectangle)o).getArea())
        return -1;
    else
        return 0;
}

public static void main(String[] args) {
    ComparableRectangle rectangle1 = new ComparableRectangle(4, 5);
    ComparableRectangle rectangle2 = new ComparableRectangle(3, 6);
    System.out.println(Max.max(rectangle1, rectangle2));
    }
}
```

12. Cloneable Interface

标记接口（Marker Interface）

标记接口是一个空接口（不包含常量或方法），用于向编译器和 JVM 表示某个类具有某些特定的属性。

A marker interface is an empty interface (does not contain constants or methods), but it is used to denote that a class possesses certain desirable properties to the compiler and the JVM.

Cloneable 接口的定义

`Cloneable` 接口定义在 `java.lang` 包中。

The `Cloneable` interface is defined in the `java.lang` package.

```
package java.lang;

public interface Cloneable {
}
```

实现 `Cloneable` 接口的类表明其对象可以被克隆。

A class that implements the `Cloneable` interface is marked cloneable:

克隆对象

实现 `Cloneable` 接口的类，其对象可以使用 `Object` 类中定义的 `clone()` 方法进行克隆，并且我们可以在我们的类中重写此方法。

Its objects can be cloned using the `clone()` method defined in the `Object` class, and we can override this method in our classes.

Cloneable 接口的示例

Java 库中的 `Calendar` 类实现了 `Cloneable` 接口。

The `Calendar` class (in the Java library) implements the `Cloneable` interface:

```
Calendar calendar = new GregorianCalendar(2022, 1, 1);
Calendar calendarCopy = (Calendar)(calendar.clone());
System.out.println("calendar == calendarCopy is " + (calendar == calendarCopy));
```

输出：

Displays:

```
calendar == calendarCopy is false
```

因为引用是不同的。

Because the references are different.

```
System.out.println("calendar.equals(calendarCopy) is " + calendar.equals(calendarCopy));
```

输出：

Displays:

```
calendar.equals(calendarCopy) is true
```

因为 `calendarCopy` 是 `calendar` 的副本。

Because the `calendarCopy` is a copy of the `calendar`.

实现 Cloneable 接口

如果尝试克隆一个未实现 `Cloneable` 接口的类的对象实例，会抛出 `CloneNotSupportedException`。

If we try to create a clone of an object instance of a class that does not implement the `Cloneable` interface, it throws `CloneNotSupportedException`.

`clone()` 方法的工作原理

`Object` 类中的 `clone()` 方法创建此对象类的一个新实例，并使用**反射技术**初始化其所有字段，其**内容与此对象的相应字段的内容完全相同。引用数据字段**的内容不会被克隆。

The `clone()` method in the `Object` class creates a new instance of the class of this object and initializes all its fields with exactly the contents of the corresponding fields of this object, as if by assignment (using a technique named **reflection**); the contents of the reference data fields are not cloned.

`clone()` 方法的返回值

`clone()` 方法返回一个需要进行类型转换的 `Object`。

The `clone()` method returns an `Object` that needs to be casted.

我们可以从 `Object` 类重写 `clone()` 方法以创建自定义克隆。

We can override the `clone()` method from the `Object` class to create custom clones.

示例：自定义克隆

假设我们有一个 `Person` 类，并希望实现自定义的克隆方法。

Let's assume we have a `Person` class and we want to implement a custom clone method.

```
public class Person implements Cloneable {
    private String name;
    private int age;

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

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    @Override
```

```
public String toString() {
    return name + ": " + age;
}

public static void main(String[] args) {
    try {
        Person person1 = new Person("John", 30);
        Person person2 = (Person) person1.clone();

        System.out.println("person1 == person2 is " + (person1 == person2)); // 输出 false
        System.out.println("person1.equals(person2) is " + person1.equals(person2)); // 输出 true

        System.out.println("person1: " + person1); // 输出 John: 30
        System.out.println("person2: " + person2); // 输出 John: 30
    } catch (CloneNotSupportedException e) {
        e.printStackTrace();
    }
}
```

结论

标记接口（如 `Cloneable`）用于向编译器和 JVM 标识某个类具有特定的属性。

Marker interfaces (like `Cloneable`) are used to denote that a class possesses certain properties to the compiler and the JVM.

实现 `Cloneable` 接口的类的对象可以使用 `clone()` 方法进行克隆，且我们可以在需要时重写此方法以提供自定义的克隆行为。

Objects of classes that implement the `Cloneable` interface can be cloned using the `clone()` method, and we can override this method to provide custom cloning behavior when needed.

```
public class House implements Cloneable, Comparable {
    private int id;
    private double area;
    private java.util.Date whenBuilt;
    public House(int id, double area) {this.id = id; this.area = area;
    whenBuilt = new java.util.Date();}

    public double getId() { return id;}
    public double getArea() { return area;}
    public java.util.Date getWhenBuilt() { return whenBuilt;}
/** Override the protected clone method defined in the Object
class, and strengthen its accessibility */
public Object clone() {
       try {
            return super.clone();
         }catch (CloneNotSupportedException ex) {
              return null;
          }
       }
/** Implement the compareTo method defined in Comparable */
public int compareTo(Object o) {
   if (area > ((House)o).area)
      return 1;
   else if (area < ((House)o).area)
      return -1;
   else
      return 0;
```

```
    }
}
```

13. Deep copying vs. Shallow copying

- **浅拷贝**：只复制对象的引用，而不复制引用的对象本身。

  Shallow Copy: Copies the object references, not the objects themselves.

```
public class House implements Cloneable {
    private int id;
    private double area;
    private Date whenBuilt;

    public House(int id, double area, Date whenBuilt) {
        this.id = id;
        this.area = area;
        this.whenBuilt = whenBuilt;
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone(); // 浅拷贝
    }

    @Override
    public String toString() {
        return "House [id=" + id + ", area=" + area + ", whenBuilt="
        + whenBuilt + "]";
    }

    public static void main(String[] args) {
        try {
            House house1 = new House(1, 1750.50, new Date());
            House house2 = (House) house1.clone(); // 浅克隆

            System.out.println("house1: " + house1);
            System.out.println("house2: " + house2);
            System.out.println("house1 == house2: " + (house1 == house2)); // false
            System.out.println("house1.whenBuilt == house2.whenBuilt: " +
                (house1.whenBuilt == house2.whenBuilt)); // true
            } catch (CloneNotSupportedException e) {
                e.printStackTrace();
            }
        }
}
```

```
public class House implements Cloneable {
    private int id;
    private double area;
    private Date whenBuilt;

    public House(int id, double area, Date whenBuilt) {
        this.id = id;
        this.area = area;
        this.whenBuilt = whenBuilt;
    }
```

```
@Override
protected Object clone() {
    try {
        House cloned = (House) super.clone();
        cloned.whenBuilt = (Date) whenBuilt.clone(); // 深拷贝
        return cloned;
    } catch (CloneNotSupportedException e) {
        return null;
    }
}

@Override
public String toString() {
    return "House [id=" + id + ", area=" + area + ",
    whenBuilt=" + whenBuilt + "]";
}

    public static void main(String[] args) {
        try {
            House house1 = new House(1, 1750.50, new Date());
            House house2 = (House) house1.clone();

            System.out.println("house1: " + house1);
            System.out.println("house2: " + house2);
            System.out.println("house1 == house2: " + (house1 == house2)); // false
            System.out.println("house1.whenBuilt == house2.whenBuilt: "
             + (house1.whenBuilt == house2.whenBuilt)); // false
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

- **深拷贝**：递归地复制对象及其引用的所有对象。

  Deep Copy: Recursively copies the objects and the objects they reference.

14. Interface vs. abstract class

接口与抽象类的比较

在 Java 中，接口和抽象类是两种不同的方式来定义抽象类型，它们有一些区别和特点。

变量

- **接口**：数据字段必须是常量（`public static final`），**不能包含实例变量。**
- **抽象类**：可以包含实例变量。

构造函数

- **接口**：**没有构造函数**，因为接口不能被实例化。
- **抽象类**：可以有构造函数，用于子类实例化时的构造过程。**constructor chain**

方法

- **接口**：方法只有签名(**signature**)，没有实现（即只有抽象方法）。
- **抽象类**：可以有抽象方法和具体方法（有方法体的方法）。

继承

- **接口**：可以继承多个接口，没有接口的根。implement interface（**There is no root for interfaces**）
- **抽象类**：只能继承一个抽象类，可以继承多个接口。extend abstract class

总结

- **接口**：用于定义抽象的行为规范，实现多态性。变量是常量，方法都是抽象的，不能被实例化。适用于多继承和解耦的场景。
- **抽象类**：用于定义抽象类别，封装通用功能，提供默认实现，可以包含实例变量。适用于代码复用和类层次结构的设计。

在选择接口还是抽象类时，要根据具体的需求和设计目标来决定使用哪种抽象类型。

15. 接口冲突 Conflicting interfaces

冲突的接口

当一个类实现了两个包含冲突信息的接口时，编译器会检测到并报错。具体情况包括：

- **常量冲突**：如果两个接口中定义了相同名称的常量，但它们的值不同。
- **方法签名冲突**：如果两个接口中定义了相同签名的方法，但它们的返回类型不同。

这种情况下，Java 编译器无法确定该使用哪个常量值或方法定义，因此会报错。

```
interface InterfaceA {
    int CONSTANT = 100;
    void doSomething();
    int conflictingMethod();
}

interface InterfaceB {
    int CONSTANT = 200; // 冲突的常量
    void doSomething();
    String conflictingMethod(); // 冲突的方法签名
}

class ConflictingClass implements InterfaceA, InterfaceB {
    // 实现时会出现编译错误，因为常量和方法签名冲突
    @Override
    public void doSomething() {
        System.out.println("Doing something");
    }

    // 无法实现 conflictingMethod()，因为返回类型冲突
    // @Override
    // public int conflictingMethod() {
    //     return 0;
    // }

    // @Override
    // public String conflictingMethod() {
    //     return "Conflict";
    // }
}
```

何时使用类或接口

选择使用类还是接口取决于所要表示的关系类型：

- **强 is-a 关系**：这是一种明确的父子关系，表示子类是父类的一种具体类型。这种关系通常使用**类继承**来建模。extends

- 例如：`Student` 类是 `Person` 类的一种具体类型，因此学生是人，这种关系应该使用类继承。

```
class Person {
    String name;

    public Person(String name) {
        this.name = name;
    }

    public void introduce() {
        System.out.println("Hi, my name is " + name);
    }
}

class Student extends Person {
    String school;

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

    @Override
    public void introduce() {
        System.out.println("Hi, my name is " + name + " and I study at " + school);
    }
}

public class Main {
    public static void main(String[] args) {
        Student student = new Student("Alice", "XYZ University");
        student.introduce(); // 输出: Hi, my name is Alice and I study at XYZ University
    }
}
```

- **弱 is-a 关系（或 is-kind-of 关系）**：这表示一个对象具有某种特性或能力，而不一定是父子关系。这种关系通常使用**接口**来建模。
  - 例如：所有字符串 (`String`) 都是可比较的 (`Comparable`)，所有日期 (`Date`) 也是可比较的。这种情况下，可以使用接口来定义这些能力。

```
interface Comparable<T> {
    int compareTo(T o);
}

class MyString implements Comparable<MyString> {
    String value;

    public MyString(String value) {
        this.value = value;
    }

    @Override
    public int compareTo(MyString other) {
        return this.value.compareTo(other.value);
    }
}
```

```
class MyDate implements Comparable<MyDate> {
    long timestamp;

    public MyDate(long timestamp) {
        this.timestamp = timestamp;
    }

    @Override
    public int compareTo(MyDate other) {
        return Long.compare(this.timestamp, other.timestamp);
    }
}

public class Main {
    public static void main(String[] args) {
        MyString str1 = new MyString("apple");
        MyString str2 = new MyString("banana");
        System.out.println(str1.compareTo(str2)); // 输出负数，因为 "apple" 小于 "banana"

        MyDate date1 = new MyDate(1627815600000L);
        MyDate date2 = new MyDate(1627815601000L);
        System.out.println(date1.compareTo(date2)); // 输出负数，因为 date1 早于 date2
    }
}
```

此外，接口还可以用来规避单一继承的限制。在 Java 中，一个类只能继承一个父类，但可以实现多个接口。这使得**接口非常适合用于定义多个不相关的功能，并允许一个类实现这些功能，从而实现多重继承的效果**。

```
// 定义飞行接口
interface Flyable {
    void fly();
}

// 定义隐身接口
interface Invisible {
    void becomeInvisible();
}

// 定义一个基类
class Person {
    String name;

    public Person(String name) {
        this.name = name;
    }

    public void introduce() {
        System.out.println("Hi, my name is " + name);
    }
}

// 定义一个超级英雄类，继承Person类并实现Flyable和Invisible接口
class Superhero extends Person implements Flyable, Invisible {
    public Superhero(String name) {
        super(name);
    }

    @Override
```

```
    public void fly() {
        System.out.println(name + " is flying!");
    }

    @Override
    public void becomeInvisible() {
        System.out.println(name + " is now invisible!");
    }
}

// 测试类
public class Main {
    public static void main(String[] args) {
        Superhero superhero = new Superhero("Superman");
        superhero.introduce();        // 输出: Hi, my name is Superman
        superhero.fly();              // 输出: Superman is flying!
        superhero.becomeInvisible();  // 输出: Superman is now invisible!
    }
}
```

总结

- **冲突的接口**：当一个类实现了两个接口，而这两个接口包含冲突的信息（例如**相同的常量但不同的值，或相同的签名但不同的返回类型的方法**）时，编译器会报错。
- **使用类还是接口**：如果要表示强 is-a 关系（明确的父子关系），应使用类继承。如果要表示弱 is-a 关系（对象具有某种特性），应使用接口。此外，如果需要多重继承，可以通过接口来实现。

16. Wrapper classes

原始数据类型和包装类

在 Java 中，原始数据类型（primitive data types）提供了更好的性能，因为它们直接存储在堆栈内存中。然而，Java 的一些数据结构（如 `ArrayList`）期望对象作为元素。因此，每个原始类型都有一个对应的包装类，这些包装类将原始数据类型包装成对象。

- **栈内存（stack）**：用于**存储方法调用和局部变量**，具有自动管理的特性，方法调用结束后自动释放。存取速度快，但大小有限。
- **堆内存（heap）**：用于**存储所有对象和数组**，生命周期由垃圾收集器管理。存取速度相对较慢，但内存大小较大。

包装类的用途

1. **在集合类中使用**：Java 的集合类（如 `ArrayList`、`HashSet`、`HashMap` 等）**只接受对象作为元素**。为了在这些集合中使用原始数据类型，需要将它们转换为相应的包装类。
2. **提供有用的方法**：包装类提供了许多有用的方法。例如，`Integer` 类提供了将字符串转换为整数的 `parseInt` 方法。
3. **表示缺失的值**：包装类可以表示缺失的值或 `null`，而原始数据类型不能表示 `null`。

包装类的特性

在 Java 中，包装类（Wrapper Classes）为原始数据类型提供了一层对象包装。它们有一些独特的特性和行为，使得它们在某些场景下非常有用。

1. 包装类没有无参构造函数 no no-arg constructor

包装类没有无参构造函数，这意味着你不能像这样实例化一个包装类对象：

```
Integer num = new Integer(); // 编译错误
```

2. 包装类的实例是不可变的 immutable

包装类的实例是不可变的。一旦创建了包装类对象，它们的内部值就不能再被改变。这种不可变性确保了包装类对象的线程安全性和一致性。

```
Integer num = new Integer(10);
// num 的值不能被改变
```

3. 重写了 `toString` 和 `equals` 方法

每个包装类都重写了 `Object` 类中的 `toString` 和 `equals` 方法。

- `toString` **方法**：返回包装对象的字符串表示形式。

  ```
  Integer num = new Integer(10);
  System.out.println(num.toString()); // 输出 "10"
  ```

- `equals` **方法**：用于比较两个包装对象的值是否相等。

  ```
  Integer num1 = new Integer(10);
  Integer num2 = new Integer(10);
  System.out.println(num1.equals(num2)); // 输出 "true"
  ```

4. 实现了 `Comparable` 接口

包装类实现了 `Comparable` 接口，因此包装类对象可以进行自然排序。`compareTo` 方法用于比较两个包装对象的值。

```
Integer num1 = new Integer(10);
Integer num2 = new Integer(20);

int result = num1.compareTo(num2);
if (result < 0) {
    System.out.println("num1 is less than num2");
} else if (result > 0) {
    System.out.println("num1 is greater than num2");
} else {
    System.out.println("num1 is equal to num2");
}
```

例子

以下是展示上述特性的代码示例：

```
public class WrapperClassExample {
    public static void main(String[] args) {
        // 无法使用无参构造函数
        // Integer num = new Integer(); // 编译错误

        // 创建包装类对象
        Integer num1 = new Integer(10);
        Integer num2 = new Integer(20);

        // 包装类对象是不可变的
        // num1 = 15; // 编译错误

        // 使用 toString 方法
        System.out.println("num1: " + num1.toString()); // 输出: num1: 10

        // 使用 equals 方法
        Integer num3 = new Integer(10);
        System.out.println("num1 equals num3: " + num1.equals(num3)); // 输出: num1 equals num3: true
```

```
        // 使用 compareTo 方法
        int comparison = num1.compareTo(num2);
        if (comparison < 0) {
            System.out.println("num1 is less than num2"); // 输出: num1 is less than num2
        } else if (comparison > 0) {
            System.out.println("num1 is greater than num2");
        } else {
            System.out.println("num1 is equal to num2");
        }
    }
}
```

小结

- **无参构造函数**：包装类没有无参构造函数。 no-arg constructors
- **不可变性**：包装类的实例是不可变的。**immutable**
- `toString` **和** `equals` **方法**：包装类重写了 `toString` 和 `equals` 方法。
- `Comparable` **接口**：包装类实现了 `Comparable` 接口，可以进行自然排序。

这些特性使得包装类在需要对象化原始数据类型时非常有用，同时也提供了一些额外的功能和方法。

17. Number class

数值包装类与抽象 `Number` 类

在 Java 中，每个数值包装类都扩展了抽象类 `Number`。这个抽象类提供了一些方法来将包装类对象转换为原始类型值。

抽象 `Number` 类的方法

`Number` 类包含以下方法，这些方法用于将对象转换为不同的原始类型值：

1. **抽象方法**：
   - `doubleValue()`: 转换为 `double` 值
   - `floatValue()`: 转换为 `float` 值
   - `intValue()`: 转换为 `int` 值
   - `longValue()`: 转换为 `long` 值
2. **非抽象方法**：
   - `byteValue()`: 返回 `(byte) intValue()`
   - `shortValue()`: 返回 `(short) intValue()`

数值包装类

每个数值包装类（如 `Integer`, `Double`, `Float`, `Long` 等）都实现了 `Number` 类中的抽象方法。

示例

以下是数值包装类及其方法的具体示例：

```
public class NumberClassExample {
    public static void main(String[] args) {
        Integer integer = new Integer(42);
        Double doubleValue = new Double(3.14);

        // 使用 Number 类的方法将 Integer 对象转换为不同的原始类型值
        System.out.println("Integer as double: " + integer.doubleValue()); // 42.0
        System.out.println("Integer as float: " + integer.floatValue()); // 42.0
        System.out.println("Integer as int: " + integer.intValue()); // 42
```

```
        System.out.println("Integer as long: " + integer.longValue()); // 42
        System.out.println("Integer as byte: " + integer.byteValue()); // 42
        System.out.println("Integer as short: " + integer.shortValue()); // 42

        // 使用 Number 类的方法将 Double 对象转换为不同的原始类型值
        System.out.println("Double as double: " + doubleValue.doubleValue()); // 3.14
        System.out.println("Double as float: " + doubleValue.floatValue()); // 3.14
        System.out.println("Double as int: " + doubleValue.intValue()); // 3
        System.out.println("Double as long: " + doubleValue.longValue()); // 3
        System.out.println("Double as byte: " + doubleValue.byteValue()); // 3
        System.out.println("Double as short: " + doubleValue.shortValue()); // 3
    }
}
```

关键点

- **抽象方法**：`Number` 类的抽象方法需要在子类中实现，这些方法用于将对象转换为特定的原始类型值。
- **非抽象方法**：`byteValue` 和 `shortValue` 方法已经在 `Number` 类中实现，它们分别调用 `(byte) intValue()` 和 `(short) intValue()`。
- **包装类实现**：每个数值包装类都实现了 `Number` 类中的抽象方法，提供具体的转换逻辑。

通过理解 `Number` 类及其方法，可以更好地理解数值包装类在 Java 中的工作方式，以及如何将包装类对象转换为不同的原始类型值。

构造器

你可以通过原始数据类型的值或表示数值的字符串来构造一个包装类对象。例如，`Integer` 和 `Double` 的构造函数如下：

```
public Integer(int value)
public Integer(String s)
public Double(double value)
public Double(String s)
```

常量

每个数值包装类都有两个常量：`MAX_VALUE` 和 `MIN_VALUE`。

- `MAX_VALUE`：表示相应原始数据类型的最大值。
- `MIN_VALUE`：对于 `Float` 和 `Double`，表示最小正浮点值。

例如：

- 最大整数：`2,147,483,647`
- 最小正浮点数：`1.4E-45`
- 最大双精度浮点数：`1.79769313486231570e+308d`

静态 `valueOf` 方法

数值包装类有一个静态方法 `valueOf(String s)`，用于**创建一个初始化为指定字符串所表示的值的新对象**。

```
Double doubleObject = Double.valueOf("12.4");
Integer integerObject = Integer.valueOf("12");
```

解析方法

每个数值包装类都有**重载**的解析方法，用于将**数字字符串解析成适当的数值**。

```
double d = Double.parseDouble("12.4");
int i = Integer.parseInt("12");
```

自动转换（装箱和拆箱）

从 JDK 1.5 开始，Java 允许原始类型和包装类之间的自动转换。

- **装箱**（Boxing）：将**原始类型转换为包装类型**，当需要对象时使用。
- **拆箱**（Unboxing）：将**包装类型转换为原始类型**，当需要原始类型时使用。

例如：

```
int primitiveInt = 10;
Integer wrapperInt = primitiveInt; // 自动装箱

int anotherPrimitiveInt = wrapperInt; // 自动拆箱
```

示例代码

以下是展示上述概念的代码示例：

```
public class WrapperClassDemo {
    public static void main(String[] args) {
        // 使用构造器创建包装类对象
        Integer intFromValue = new Integer(42);
        Integer intFromString = new Integer("42");
        Double doubleFromValue = new Double(3.14);
        Double doubleFromString = new Double("3.14");

        // 使用 MAX_VALUE 和 MIN_VALUE 常量
        System.out.println("Max Integer: " + Integer.MAX_VALUE);
        System.out.println("Min Positive Float: " + Float.MIN_VALUE);

        // 使用静态 valueOf 方法
        Double doubleObject = Double.valueOf("12.4");
        Integer integerObject = Integer.valueOf("12");

        // 使用解析方法
        double parsedDouble = Double.parseDouble("12.4");
        int parsedInt = Integer.parseInt("12");

        // 自动装箱和拆箱
        int primitiveInt = 10;
        Integer wrapperInt = primitiveInt; // 自动装箱
        int anotherPrimitiveInt = wrapperInt; // 自动拆箱

        // 输出结果
        System.out.println("intFromValue: " + intFromValue);
        System.out.println("intFromString: " + intFromString);
        System.out.println("doubleFromValue: " + doubleFromValue);
        System.out.println("doubleFromString: " + doubleFromString);
        System.out.println("doubleObject: " + doubleObject);
        System.out.println("integerObject: " + integerObject);
        System.out.println("parsedDouble: " + parsedDouble);
        System.out.println("parsedInt

```java
        System.out.println("parsedInt: " + parsedInt);
        System.out.println("primitiveInt: " + primitiveInt);
        System.out.println("wrapperInt: " + wrapperInt);
        System.out.println("anotherPrimitiveInt: " + anotherPrimitiveInt);
    }
}
```
```

总结

- **数值包装类**：提供了对原始类型的对象包装及相关操作。
- **构造方法**：可以通过原始类型的值或字符串来构造包装类对象。
- **常量**：每个数值包装类都有 `MAX_VALUE` 和 `MIN_VALUE` 常量。
- **静态方法**：使用 `valueOf` 方法创建新的包装类对象。
- **解析方法**：将数字字符串解析成原始类型值。
- **自动转换**：Java 支持原始类型与包装类之间的自动转换（装箱和拆箱）。

这些特性使得数值包装类在处理数据类型转换、集合操作等方面非常方便和灵活。

17. Arrays are objects

数组是对象

在 Java 中，数组是对象，这意味着数组具有类对象的一些特性和行为。以下是一些关键点和示例来解释这一点。

数组是对象

- **数组是 `Object` 类的实例**：你可以使用 `instanceof` 运算符来检查数组是否是 `Object` 类的实例。

```
new int[10] instanceof Object // true
```

- **数组类型的继承关系**：如果 `A` 是 `B` 的子类，那么 `A[]` 的每个实例也是 `B[]` 的实例。

```
new GregorianCalendar[10] instanceof Calendar[] // true
new Calendar[10] instanceof Object[] // true
new Calendar[10] instanceof Object // true
```

- **数组类型的兼容性**：虽然一个 `int` 值可以分配给一个 `double` 类型的变量，但 `int[]` 和 `double[]` 是两种不兼容的类型。

```
double[] a = new int[10]; // 编译错误
```

排序对象数组

Java 提供了一个静态的 `sort` 方法来排序 `java.util.Arrays` 类中的对象数组，该方法使用 `Comparable` 接口。

```
java.util.Arrays.sort(intArray);
```

排序的对象是 `Comparable` 接口的实例，并使用 `compareTo` 方法进行比较。

示例代码

以下是一个示例代码，展示如何对对象数组进行排序：

```
import java.util.Arrays;

public class GenericSort {
    public static void main(String[] args) {
        Integer[] intArray = {new Integer(2), new Integer(4), new Integer(3)};
        sort(intArray); // 或者使用 Arrays.sort(intArray);
        printList(intArray);
    }

    public static void sort(Object[] list) {
        Object currentMax;
        int currentMaxIndex;
```

```
    for (int i = list.length - 1; i >= 1; i--) {
        currentMax = list[i];
        currentMaxIndex = i;
        // 在 list[0..i] 中找到最大值
        for (int j = i - 1; j >= 0; j--) {
            if (((Comparable)currentMax).compareTo(list[j]) < 0) {
                currentMax = list[j];
                currentMaxIndex = j;
            }
        }
        // 交换位置
        list[currentMaxIndex] = list[i];
        list[i] = currentMax;
    }
}

public static void printList(Object[] list) {
    for (int i = 0; i < list.length; i++) {
        System.out.print(list[i] + " ");
    }
  }
}
```

解释

- `sort` **方法**：使用选择排序算法对对象数组进行排序。
  - **选择排序**：对于每个位置 `i`，从数组中找到 `0` 到 `i` 范围内的最大值，并将其与当前位置的值进行交换。
  - **比较**：使用 `Comparable` 接口的 `compareTo` 方法来比较对象。
- `printList` **方法**：打印数组中的所有元素。

总结

- 数组在 Java 中是对象，并具有对象的特性。
- 数组类型之间有一定的继承关系。
- 使用 `java.util.Arrays.sort` 方法可以排序实现了 `Comparable` 接口的对象数组。
- 在编写排序方法时，需要注意类型的兼容性和使用合适的比较方法。

通过理解这些概念，可以更好地利用数组和对象在 Java 中的特性来编写高效和灵活的代码。

18.BigInteger and BigDecimal

BigInteger 和 BigDecimal

`BigInteger` 和 `BigDecimal` 是 Java 中用于处理非常大的整数和高精度浮点数的类，位于 `java.math` 包中。

BigInteger

- **表示任意大小的整数**：可以表示超出 `long` 类型范围的整数。
- **不可变**：一旦创建，内部值不能改变。
- **继承和接口**：继承自 `Number` 类，实现了 `Comparable` 接口。

BigDecimal

- **高精度浮点数**：没有精度限制，只要它是有限的（终止的）。
- **不可变**：一旦创建，内部值不能改变。
- **继承和接口**：继承自 `Number` 类，实现了 `Comparable` 接口。

示例代码

BigInteger 示例

以下是使用 `BigInteger` 类进行大整数运算的示例：

```
import java.math.BigInteger;

public class BigIntegerExample {
    public static void main(String[] args) {
        BigInteger a = new BigInteger("92233720368547758O7");
        BigInteger b = new BigInteger("2");
        BigInteger c = a.multiply(b); // 92233720368547758O7 * 2
        System.out.println(c); // 输出 184467440737095516l4
    }
}
```

BigDecimal 示例

以下是使用 `BigDecimal` 类进行高精度浮点运算的示例：

```
import java.math.BigDecimal;

public class BigDecimalExample {
    public static void main(String[] args) {
        BigDecimal a = new BigDecimal(1.0);
        BigDecimal b = new BigDecimal(3);
        BigDecimal c = a.divide(b, 20, BigDecimal.ROUND_UP); // 设置精度为20位
        System.out.println(c); // 输出 0.33333333333333333334
    }
}
```

计算大阶乘

以下是使用 `BigInteger` 计算大阶乘的示例：

```
import java.math.BigInteger;

public class LargeFactorial {
    public static void main(String[] args) {
        System.out.println("50! is \\n" + factorial(50));
    }

    public static BigInteger factorial(long n) {
        BigInteger result = BigInteger.ONE;
        for (int i = 1; i <= n; i++) {
            result = result.multiply(new BigInteger(i + ""));
        }
        return result;
    }
}
```

解释

- **不可变性**：`BigInteger` 和 `BigDecimal` 的对象一旦创建，内部值不可更改。这保证了线程安全性和数据一致性。
- **高精度运算**：这些类提供了丰富的方法进行精确的数学运算，例如加法、减法、乘法、除法和求幂。
- **扩展性**：由于它们继承了 `Number` 类，并实现了 `Comparable` 接口，可以用于排序和数值转换。

总结

- **BigInteger** 和 **BigDecimal** 是处理大数和高精度浮点数的强大工具，适用于需要精确计算的场景。
- 通过使用这些类，可以避免由于原始数据类型范围限制而导致的计算错误。
- 它们的不可变性和丰富的操作方法使得在多线程和精确计算场景中尤为有用。

Lecture 4 Generics

1. Generics is the capability to parameterize types

Java中的泛型（Generics）

泛型（Generics）允许你定义带有类型参数（type parameters）的类、接口和方法。这一特性通过让编译器在编译时而不是在运行时检测错误，提高了代码的重用性和类型安全性。

泛型的主要特点

1. **参数化类型（Parameterize Types）**：
   - 泛型允许你创建操作泛型类型（generic types）的类或方法。这些类型在类或方法实例化或调用时由用户指定。
   - 例如，你可以定义一个泛型栈类（generic stack class），可以存储任何类型的元素。
2. **编译时类型检查（Compile-Time Type Checking）**：
   - 泛型使编译器能够强制执行类型安全。如果你尝试使用不兼容的类型，编译器将生成错误，从而防止运行时类型错误。
   - 例如，一个`Stack<String>`将只接受`String`对象。
3. **代码重用（Code Reuse）**：
   - 泛型通过允许你为各种数据结构或算法编写单一实现来促进代码重用。
   - 例如，你可以编写一个泛型栈（generic stack）的单一实现，并为不同的数据类型（如`String`、`Integer`等）重用它。
4. **消除强制类型转换（Eliminates Casting）**：
   - 没有泛型，你在从集合中检索对象时需要进行类型转换。使用泛型后，不需要类型转换，因为编译器知道对象的类型。
   - 示例：

```
// 没有泛型（JDK 1.4及之前）
ArrayList list = new ArrayList();
list.add(new Integer(1));
Integer i = (Integer) list.get(0);

// 使用泛型（JDK 1.5及之后）
ArrayList<Integer> list = new ArrayList<>();
list.add(1);
Integer i = list.get(0);
```

5. **不支持原始类型（No Primitive Types）**：
   - 泛型类型必须是引用类型（reference types）。你不能使用原始类型（primitive types）如`int`、`double`或`char`作为泛型类型参数。
   - 示例：

```
// 错误用法：编译错误
ArrayList<int> intList = new ArrayList<>();

// 使用包装类的正确用法
ArrayList<Integer> intList = new ArrayList<>();
intList.add(1); // 自动装箱（Autoboxing）
```

示例：泛型栈类（Generic Stack Class）

以下是一个泛型栈类的示例：

```
public class GenericStack<T> {
    private java.util.ArrayList<T> list = new java.util.ArrayList<>();

    public int getSize() {
        return list.size();
    }

    public T peek() {
        return list.get(getSize() - 1);
    }

    public void push(T o) {
        list.add(o);
    }

    public T pop() {
        T o = list.get(getSize() - 1);
        list.remove(getSize() - 1);
        return o;
    }

    public boolean isEmpty() {
        return list.isEmpty();
    }
}

public class TestGenericStack {
    public static void main(String[] args) {
        // 存储字符串的栈（Stack for Strings）
        GenericStack<String> stringStack = new GenericStack<>();
        stringStack.push("Hello");
        stringStack.push("World");
        System.out.println(stringStack.pop()); // 输出：World

        // 存储整数的栈（Stack for Integers）
        GenericStack<Integer> integerStack = new GenericStack<>();
        integerStack.push(1);
        integerStack.push(2);
        System.out.println(integerStack.pop()); // 输出：2
    }
}
```

泛型的优点

1. **类型安全（Type Safety）**：
   - 泛型通过确保你只使用兼容类型来提供类型安全。编译器在编译时检查这一点，从而减少运行时错误。
2. **消除类型转换（Eliminates Casting）**：
   - 泛型消除了从集合中检索元素时的显式类型转换，使代码更清晰且不易出错。
3. **提高代码重用性（Improved Code Reuse）**：
   - 使用泛型，你可以编写更灵活和可重用的代码。例如，一个单一的泛型类可以处理不同类型。
4. **增强可读性（Enhanced Readability）**：
   - 使用泛型的代码更加可读和易于理解，因为它清楚地表明了操作的类型。

结论

Java中的泛型提供了一种编写灵活、类型安全和可重用代码的强大方式。通过允许你参数化类型，泛型有助于确保程序没有类型相关的错误，减少类型转换并增强代码的可读性。

2. Using Generics

通用静态方法

一个通用的静态方法允许对不同类型进行类型安全的操作，而无需指定具体类型。
A generic static method allows for type-safe operations on different types without specifying a concrete type.

示例：
Example:

```
public class GenericMethods1 {
    public static <E> void print(E[] list) {
        for (E element : list) {
            System.out.print(element + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        String[] s3 = {"Hello", "again"};
        GenericMethods1.<String>print(s3); // 明确的类型指定
        print(s3); // 类型推断
    }
}
```

有界通用类型

你可以使用 `extends` 关键字将通用类型限制为另一个类型的子类型。
You can restrict a generic type to be a subtype of another type using the `extends` keyword.

示例：
Example:

```
public class GenericMethods2 {
    public static <E extends GeometricObject> boolean equalArea(E object1, E object2) {
        return object1.getArea() == object2.getArea();
    }

    public static void main(String[] args) {
        Rectangle rectangle = new Rectangle(2, 2);
        Circle circle = new Circle(2);
        System.out.println("Same area? " + equalArea(rectangle, circle));
    }
}

// 假设 GeometricObject、Circle 和 Rectangle 被正确定义并具有 getArea 方法
// Assume GeometricObject, Circle, and Rectangle are properly defined with getArea method
```

排序对象数组

一个通用的方法可以对实现 `Comparable` 接口的对象数组进行排序。
A generic method can sort an array of objects that implement the `Comparable` interface.

示例：
Example:

```
public class GenericSelectionSort {
    public static <E extends Comparable<E>> void genericSelectionSort(E[] list) {
        for (int i = 0; i < list.length - 1; i++) {
            E currentMin = list[i];
            int currentMinIndex = i;

            for (int j = i + 1; j < list.length; j++) {
                if (currentMin.compareTo(list[j]) > 0) {
                    currentMin = list[j];
                    currentMinIndex = j;
                }
            }

            if (currentMinIndex != i) {
                list[currentMinIndex] = list[i];
                list[i] = currentMin;
            }
        }
    }

    public static void main(String[] args) {
        Integer[] intArray = {2, 4, 3};
        GenericSelectionSort.<Integer>genericSelectionSort(intArray);
        System.out.print("Sorted Integer objects: ");
        printList(intArray);

        Double[] doubleArray = {3.4, 1.3};
        GenericSelectionSort.<Double>genericSelectionSort(doubleArray);
        System.out.print("Sorted Double objects: ");
        printList(doubleArray);
    }

    public static void printList(Object[] list) {
        for (Object element : list) {
            System.out.print(element + " ");
        }
        System.out.println();
    }
}
```

原始类型和向后兼容性 Raw Type and Backward Compatibility

使用原始类型可能导致不安全的操作。

A generic class or interface used without specifying a concrete type, called a raw type, enables backward compatibility with earlier versions of Java

Note: Backward compatibility: a technology is backward compatible if the input developed for an older technology can work with the newer technology

在Java中使用未指定具体类型的泛型类或接口，即所谓的原始类型（raw type），可以实现向后兼容性，使得早期版本的Java程序仍然可执行。

在Java JDK 1.5及更高版本中，未指定具体类型的原始类型，比如 `ArrayList list = new ArrayList();`，大致相当于指定了一个泛型类型为 `Object` 的ArrayList，即 `ArrayList<Object> list = new ArrayList<Object>();`。

因此，通过使用原始类型，Java可以保证旧版本的程序仍然可以在新版本中运行，这体现了向后兼容性的概念：即旧版本技术开发的输入可以与新版本技术兼容，使得旧版本的代码在新版本环境中仍然可以工作。

raw type是不安全的
Using
raw types can lead to unsafe operations.

不安全的原始类型使用示例：
Example of unsafe raw type usage:

```
public class Unsafe {
    public static Comparable max1(Comparable o1, Comparable o2) {
        if (o1.compareTo(o2) > 0) return o1;
        else return o2;
    }

    public static void main(String[] args) {
        System.out.println(max1("Welcome", 23)); // 运行时错误
    }
}
```

使用泛型的安全版本：
Safe version with generics:

```
public class Safe {
    public static <E extends Comparable<E>> E max2(E o1, E o2) {
        if (o1.compareTo(o2) > 0) return o1;
        else return o2;
    }

    public static void main(String[] args) {
        // System.out.println(max2("Welcome", 23)); // 编译错误
        System.out.println(max2("Welcome", "Hello")); // 正常工作
    }
}
```

通配符

通配符用于指定泛型类型的范围。
Wildcards are used to specify a range of types in generics.

带有上界通配符的示例：
Example with
bounded wildcard:<? extends Number>

```
public class WildCardDemo1B {
    public static double max(GenericStack<? extends Number> stack) {
        double max = stack.pop().doubleValue();
        while (!stack.isEmpty()) {
            double value = stack.pop().doubleValue();
            if (value > max) max = value;
        }
        return max;
    }

    public static void main(String[] args) {
        GenericStack<Integer> intStack = new GenericStack<>();
        intStack.push(1);
        intStack.push(2);
        intStack.push(-2);
```

```
        System.out.print("The max number is " + max(intStack)); // 正常工作
    }
}
```

带有无界通配符的示例：
Example with
unbounded wildcard:<?>

```
public class WildCardDemo2 {
    public static void print(GenericStack<?> stack) {
        while (!stack.isEmpty()) {
            System.out.print(stack.pop() + " ");
        }
    }

    public static void main(String[] args) {
        GenericStack<Integer> intStack = new GenericStack<>();
        intStack.push(1);
        intStack.push(2);
        intStack.push(-2);
        print(intStack); // 输出: -2 2 1
    }
}
```

带有下界通配符的示例：
Example with
lower-bounded wildcard: <? super T>

```
public class WildCardDemo4 {
    public static <T> void add(GenericStack<T> stack1, GenericStack<? super T> stack2) {
        while (!stack1.isEmpty()) {
            stack2.push(stack1.pop());
        }
    }

    public static void print(GenericStack<?> stack) {
        while (!stack.isEmpty()) {
            System.out.print(stack.pop() + " ");
        }
    }

    public static void main(String[] args) {
        GenericStack<String> stack1 = new GenericStack<>();
        GenericStack<Object> stack2 = new GenericStack<>();
        stack2.push("Java");
        stack2.push(2);
        stack1.push("Sun");
        add(stack1, stack2);
        print(stack2); // 输出: Sun 2 Java
    }
}
```

这些示例展示了Java中泛型的灵活性和强大功能，使得跨多种类型的操作能够保持类型安全，同时保持代码的可重用性和可读性。
These examples illustrate the flexibility and power of generics in Java, enabling type-safe operations across a variety of types while maintaining code reusability and readability.

3. Restriction on Generics

Erasure and Restrictions on Generics

泛型在Java中的实现采用了一种称为类型擦除的方法：编译器在编译代码时使用泛型类型信息，但之后会将其擦除。这意味着在生成的字节码中，泛型类型信息将被替换为其原始类型或边界类型。

例如，考虑以下的泛型类声明：

```
public class Box<T> {
    private T value;
    // 省略其他代码
}
```

在编译之后，所有的`T`将被替换为其边界类型或者`Object`。因此，`Box<String>`和`Box<Integer>`在字节码层面上将是相同的。

这种方法使得泛型代码能够与使用原始类型的旧代码向后兼容。在早期版本的Java中，没有泛型，因此所有的集合类都使用原始类型。为了确保新的泛型代码可以与旧代码一起使用，编译器会将泛型类型擦除，并将其替换为原始类型。这意味着即使你在新代码中使用了泛型类型，也可以将其与旧代码一起使用，而无需进行修改。

但是，类型擦除也带来了一些限制：

1. **无法创建泛型数组**：由于类型擦除，泛型数组的创建是不允许的。例如，以下代码将会产生编译错误：

```
T[] array = new T[10];
```

2. **泛型类中不允许出现静态上下文**：因为在运行时所有的泛型类实例都共属于同一个runtime class 而静态上下文是所有泛型类实例共享的 泛型类实例的类型不同自然不能够共享一个静态方法或者变量

```
public class Example<T> {
    private static T staticField; // 非法
    public static void staticMethod(T arg) { // 非法
        // 静态上下文中不能引用泛型类型参数
    }
}
```

3. **异常类不能是泛型的**：Java的异常处理机制依赖于在运行时捕获并匹配特定类型的异常。由于类型擦除，运行时无法知道异常的泛型类型，因此异常类不能是泛型的。

The JVM has to check the exception thrown from the try clause to see
if it matches the type specified in a catch clause.
This is impossible, because the type information is not present at runtime.

```
// 非法的异常类声明
public class MyException<T> extends Exception {
    // ...
}
```

总的来说，尽管泛型在Java中提供了强大的类型安全性和灵活性，但由于类型擦除带来的限制，需要在使用时注意一些约束和规则。

Lecture5 Data structure & collection

1. Data structure

数据结构或集合是以某种方式组织的数据集合
A data structure or collection is a collection of data organized in some fashion

不仅存储数据，还支持访问和操作数据的功能
Not only stores data but also supports operations for accessing and manipulating the data

选择适合的数据结构和算法是开发高性能软件的关键之一
Choosing the best data structures and algorithms for a particular task is one of the keys to developing high-performance software

在面向对象编程中，数据结构也称为容器，是存储其他对象的对象，这些对象被称为元素
In object-oriented thinking, a data structure, also known as a container, is an object that stores other objects, referred to as elements

Java 集合框架层次结构

Java 提供了几种数据结构来高效地组织和操作数据，通常称为 Java 集合框架
Java provides several data structures that can be used to organize and manipulate data efficiently, commonly known as Java Collections Framework

Java 集合框架支持两种类型的容器：一种用于存储元素集合，称为集合
The Java Collections Framework supports two types of containers: one for storing a collection of elements, simply called a collection

列表存储有序的元素集合
Lists store an ordered collection of elements

集合存储不重复元素的组
Sets store a group of nonduplicate elements

堆栈存储以后进先出方式处理的对象
Stacks store objects that are processed in a last-in, first-out fashion

队列存储以先进先出方式处理的对象
Queues store objects that are processed in a first-in, first-out fashion

优先队列存储按优先级顺序处理的对象
PriorityQueues store objects that are processed in the order of their priorities

另一种用于存储键/值对的容器，称为映射
One for storing key/value pairs, called a map

注：在 Python 中，这称为字典
Note: This is called a dictionary in Python

Java 集合框架层次结构

Java 集合框架中定义的所有接口和类都在 java.util 包中
All the interfaces and classes defined in the Java Collections Framework are grouped in the java.util package

Java 集合框架的设计是使用接口、抽象类和具体类的一个优秀例子
The design of the Java Collections Framework is an excellent example of using interfaces, abstract classes, and concrete classes

接口定义了框架/通用 API
The interfaces define the framework/general API

抽象类提供部分实现
The abstract classes provide partial implementation

提供一个部分实现接口的抽象类使用户编写专门的容器代码变得方便
Providing an abstract class that partially implements an interface makes it convenient for the user to write the code for the specialized containers

出于方便，提供了 AbstractCollection（因此它被称为便捷抽象类）
AbstractCollection is provided for convenience (for this reason, it is called a convenience abstract class)

具体类使用具体的数据结构实现接口
The concrete classes implement the interfaces with concrete data structures

Collection 接口

Collection 接口是操作对象集合的根接口
The Collection interface is the root interface for manipulating a collection of objects

AbstractCollection 抽象类为 Collection 接口提供了部分实现（除 add、size 和 iterator 方法外的所有方法）
The AbstractCollection class provides partial implementation for the Collection interface (all the methods in Collection

except the add, size, and iterator methods)

注：Collection 接口实现了 Iterable 接口
Note: The Collection interface implements the Iterable interface

我们可以获取一个 Iterator 对象来遍历集合中的元素
We can obtain an Iterator object for traversing elements in the collection

还可以用于 for-each 循环
Also used by for-each loops

示例代码

```
import java.util.*;

public class TestCollection {
    public static void main(String[] args) {
        ArrayList<String> collection1 = new ArrayList<>();
        collection1.add("New York"); // add
        collection1.add("Atlanta");
        collection1.add("Dallas");
        collection1.add("Madison");
        System.out.println("A list of cities in collection1:");
        System.out.println(collection1);

        // the Collection interface's contains method
        System.out.println("\\nIs Dallas in collection1? "
            + collection1.contains("Dallas")); // contains

        // the Collection interface's remove method
        collection1.remove("Dallas"); // remove

        // the Collection interface's size method
        System.out.println("\\n" + collection1.size() + // size
            " cities are in collection1 now");

        Collection<String> collection2 = new ArrayList<>();
        collection2.add("Seattle");
        collection2.add("Portland");
        System.out.println("\\nA list of cities in collection2:");
        System.out.println(collection2);

        ArrayList<String> c1 = (ArrayList<String>) (collection1.clone()); // clone
        c1.addAll(collection2); // addAll
        System.out.println("\\nCities in collection1 or collection2:");
        System.out.println(c1);

        c1 = (ArrayList<String>) (collection1.clone());
        c1.retainAll(collection2); // retainAll
        System.out.print("\\nCities in collection1 and collection2:");
        System.out.println(c1);

        c1 = (ArrayList<String>) (collection1.clone());
        c1.removeAll(collection2); // removeAll
        System.out.print("\\nCities in collection1, but not in 2: ");
        System.out.println(c1);
    }
}
```

输出

```
A list of cities in collection1:
[New York, Atlanta, Dallas, Madison]

Is Dallas in collection1? true

3 cities are in collection1 now

A list of cities in collection2:
[Seattle, Portland]

Cities in collection1 or collection2:
[New York, Atlanta, Madison, Seattle, Portland]

Cities in collection1 and collection2:[]
Cities in collection1, but not in 2: [New York, Atlanta, Madison]
```

集合框架层次结构

Java 集合框架中的所有具体类都实现了 java.lang.Cloneable 和 java.io.Serializable 接口，但 java.util.PriorityQueue 没有实现 Cloneable 接口
All the concrete classes in the Java Collections Framework implement the java.lang.Cloneable and java.io.Serializable interfaces except that java.util.PriorityQueue does not implement the Cloneable interface

Collection 接口中的某些方法无法在具体子类中实现（例如，只读集合不能添加或删除元素）
Some of the methods in the Collection interface cannot be implemented in the concrete subclass (e.g., the read-only collections cannot add or remove)

在这种情况下，该方法会抛出 `java.lang.UnsupportedOperationException`，如下所示：
In this case, the method would throw java.lang.UnsupportedOperationException, like this:

```
public void someMethod() {
   throw new UnsupportedOperationException("Method not supported");
}
```

2. Iterator

补充：接口可以继承接口 从而实现功能的扩展 类implements接口 接口extends 接口 类 extends 类 一个类可以implements 多个接口

在 Java 中，接口是可以继承接口的。接口继承接口的意思是，一个接口可以扩展另一个接口，从而继承另一个接口中的抽象方法和常量。这种继承机制允许你创建更复杂、更具体的接口，同时保持代码的组织和可维护性。

示例解释

假设我们有一个基本的接口 `Animal`，它定义了一些基本的行为：

```
public interface Animal {
   void eat();
   void sleep();
}
```

现在，我们想创建一个更具体的接口 `Bird`，它不仅具有 `Animal` 的所有行为，还增加了飞行的能力：

```
public interface Bird extends Animal {
   void fly();
}
```

在这个例子中，`Bird` 接口继承了 `Animal` 接口。这意味着任何实现 `Bird` 接口的类都必须实现 `Animal` 接口中的所有方法（`eat` 和 `sleep`）以及 `Bird` 接口中新增加的方法（`fly`）。

实现类示例

现在，我们可以创建一个实现 `Bird` 接口的类，比如 `Eagle`：

```
public class Eagle implements Bird {
    @Override
    public void eat() {
        System.out.println("Eagle is eating.");
    }

    @Override
    public void sleep() {
        System.out.println("Eagle is sleeping.");
    }

    @Override
    public void fly() {
        System.out.println("Eagle is flying.");
    }
}
```

`Eagle` 类实现了 `Bird` 接口，这意味着它必须实现 `Animal` 接口中的 `eat` 和 `sleep` 方法以及 `Bird` 接口中的 `fly` 方法。

多接口继承

一个接口还可以继承多个接口。这允许一个接口组合多个接口的行为：

```
public interface Swimmer {
    void swim();
}

public interface Amphibian extends Animal, Swimmer {
    void liveOnLandAndWater();
}
```

在这个例子中，`Amphibian` 接口继承了 `Animal` 和 `Swimmer` 接口。这意味着任何实现 `Amphibian` 接口的类都必须实现 `Animal` 接口中的 `eat` 和 `sleep` 方法，以及 `Swimmer` 接口中的 `swim` 方法和 `Amphibian` 接口中新增加的 `liveOnLandAndWater` 方法。

总结

接口继承接口的机制允许你创建层次化的接口结构，使得代码更加模块化和易于扩展。这种设计方式有助于提高代码的可读性和可维护性。

迭代器

每个集合都是可迭代的
Each collection is Iterable

迭代器是一种经典的设计模式，用于遍历数据结构，而不需要暴露数据在数据结构中的存储细节
Iterator is a classic design pattern for walking through a data structure without having to expose the details of how data is stored in the data structure

也用于 for-each 循环：
Also used in for-each loops:

```
for (String element : collection)
    System.out.print(element + " ");
```

Collection 接口继承了 Iterable 接口
The Collection interface extends the Iterable interface

你可以通过 Iterable 接口中的 iterator() 方法获得一个集合的迭代器对象，来遍历集合中的所有元素
You can obtain a collection Iterator object to traverse all the elements in the collection with the iterator() method in the Iterable interface which returns an instance of Iterator

Iterable 接口定义了 iterator 方法，该方法返回一个迭代器
The Iterable interface defines the iterator method, which returns an Iterator

iterable 接口中有一个iterator()方法还有一个foreach()方法 iterator()方法返回一个iterator对象 iterator接口中有三个hasnext next 还有remove方法

示例代码

```
import java.util.*;

public class TestIterator {
    public static void main(String[] args) {
        Collection<String> collection = new ArrayList<>();
        collection.add("New York");
        collection.add("Atlanta");
        collection.add("Dallas");
        collection.add("Madison");

        Iterator<String> iterator = collection.iterator();
        while (iterator.hasNext()) {
            System.out.print(iterator.next().toUpperCase() + " ");
        }
        System.out.println();
    }
}
```

输出

```
NEW YORK ATLANTA DALLAS MADISON
```

解释

迭代器模式用于遍历集合而无需了解底层实现细节
The iterator pattern is used to traverse collections without knowing the underlying implementation details.

在上述代码中，我们创建了一个字符串集合并添加了一些城市名称
In the code above, we create a collection of strings and add some city names.

然后，我们获取该集合的迭代器对象，并使用它遍历集合中的每个元素
We then obtain an iterator for this collection and use it to traverse each element in the collection.

`iterator()` **方法返回一个迭代器，该迭代器提供了** `hasNext()` **和** `next()` **方法**
The
`iterator()` method returns an iterator which provides the `hasNext()` and `next()` methods.

`hasNext()` **方法检查集合中是否还有未遍历的元素**
The
`hasNext()` method checks if there are more elements to iterate over.

`next()` **方法返回下一个元素**
The
`next()` method returns the next element.

在循环中，我们将每个元素转换为大写并打印出来
In the loop, we convert each element to uppercase and print it.

这种方法使我们能够以一致和简洁的方式处理集合中的元素，而不必关心其底层实现细节。

3.ListIterator

ListIterator 示例

`ListIterator` 是 `Iterator` 的一个子接口，提供了更丰富的方法来遍历列表和修改列表中的元素。以下是使用 `ListIterator` 的例子，展示如何使用 `nextIndex()`、`previousIndex()` 和 `add(element)` 方法：

```
import java.util.*;

public class TestListIterator {
    public static void main(String[] args) {
        // 创建一个 ArrayList 并添加一些元素
        List<String> list = new ArrayList<>();
        list.add("New York");
        list.add("Atlanta");
        list.add("Dallas");
        list.add("Madison");

        // 获取 ListIterator 对象
        ListIterator<String> listIterator = list.listIterator();

        // 使用 nextIndex() 方法遍历元素并输出其下一个元素的索引
        System.out.println("Using nextIndex():");
        while (listIterator.hasNext()) {
            String element = listIterator.next();
            System.out.println("Element: " + element + ", Next index: " + listIterator.nextIndex());
        }

        // 使用 previousIndex() 方法遍历元素并输出其上一个元素的索引
        System.out.println("\\nUsing previousIndex():");
        while (listIterator.hasPrevious()) {
            String element = listIterator.previous();
            System.out.println("Element: " + element + ", Previous index: " + listIterator.previousIndex());
        }

        // 使用 add() 方法在特定位置插入新元素
        System.out.println("\\nAdding elements using add() method:");
        listIterator = list.listIterator();  // 重置 ListIterator
        while (listIterator.hasNext()) {
            String element = listIterator.next();
            if ("Dallas".equals(element)) {
                listIterator.add("Houston");  // 在 "Dallas" 前插入 "Houston"
            }
            System.out.println("Element: " + element);
        }

        // 输出修改后的列表
        System.out.println("\\nModified list:");
        for (String city : list) {
            System.out.println(city);
        }
    }
}
```

解释

1. **创建和初始化列表**
   - 我们创建了一个 `ArrayList` 并添加了一些城市名称。
2. **获取 `ListIterator` 对象**
   - 使用 `list.listIterator()` 方法获取 `ListIterator` 对象。
3. **使用 `nextIndex()` 方法遍历列表**
   - 使用 `hasNext()` 和 `next()` 方法遍历列表，并在每次迭代中输出当前元素及其下一个元素的索引。
4. **使用 `previousIndex()` 方法逆向遍历列表**
   - 使用 `hasPrevious()` 和 `previous()` 方法逆向遍历列表，并在每次迭代中输出当前元素及其上一个元素的索引。
5. **使用 `add()` 方法插入新元素**
   - 使用 `add()` 方法在遇到 "Dallas" 元素时，在其前面插入 "Houston"。
6. **输出修改后的列表**
   - 最后，遍历列表并输出所有元素，显示插入新元素后的列表。

输出

```
Using nextIndex():
Element: New York, Next index: 1
Element: Atlanta, Next index: 2
Element: Dallas, Next index: 3
Element: Madison, Next index: 4

Using previousIndex():
Element: Madison, Previous index: 2
Element: Dallas, Previous index: 1
Element: Atlanta, Previous index: 0
Element: New York, Previous index: -1

Adding elements using add() method:
Element: New York
Element: Atlanta
Element: Dallas
Element: Madison

Modified list:
New York
Atlanta
Houston
Dallas
Madison
```

通过这个例子，你可以看到如何使用 `ListIterator` 遍历列表，获取元素的索引，并在特定位置插入新元素。这展示了 `ListIterator` 在处理列表时的强大功能和灵活性。

4.array list vs. linked list

这个例子演示了如何使用 `ArrayList` 和 `LinkedList`。具体来说，示例将展示如何在 `ArrayList` 中插入元素，如何将 `ArrayList` 转换为 `LinkedList`，以及如何在 `LinkedList` 中插入和删除元素，并前后遍历列表。

示例说明

1. **创建并填充 ArrayList**
   - 创建一个 `ArrayList` 并填充一些数字。
   - 在指定位置插入新元素。

2. **转换为 LinkedList 并进行操作**
    - 从 `ArrayList` 创建一个 `LinkedList`。
    - 在 `LinkedList` 中插入和删除元素。
3. **遍历 LinkedList**
    - 使用 `ListIterator` 前向和后向遍历列表。

示例代码

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.ListIterator;

public class ListExample {
    public static void main(String[] args) {
        // 创建并填充 ArrayList
        ArrayList<Integer> arrayList = new ArrayList<>();
        arrayList.add(1);
        arrayList.add(2);
        arrayList.add(3);
        arrayList.add(4);
        arrayList.add(5);

        // 在指定位置插入新元素
        arrayList.add(2, 99); // 在索引2处插入99
        System.out.println("ArrayList: " + arrayList);

        // 从 ArrayList 创建 LinkedList
        LinkedList<Integer> linkedList = new LinkedList<>(arrayList);
        System.out.println("LinkedList: " + linkedList);

        // 在 LinkedList 中插入和删除元素
        linkedList.addFirst(0); // 在开头插入0
        linkedList.addLast(100); // 在末尾插入100
        linkedList.remove(2); // 删除索引2处的元素
        System.out.println("LinkedList after insertion and deletion: " + linkedList);

        // 使用 ListIterator 前向遍历列表
        ListIterator<Integer> iterator = linkedList.listIterator();
        System.out.print("Forward traversal: ");
        while (iterator.hasNext()) {
            System.out.print(iterator.next() + " ");
        }
        System.out.println();

        // 使用 ListIterator 后向遍历列表
        System.out.print("Backward traversal: ");
        while (iterator.hasPrevious()) {
            System.out.print(iterator.previous() + " ");
        }
        System.out.println();
    }
}
```

输出

```
ArrayList: [1, 2, 99, 3, 4, 5]
LinkedList: [1, 2, 99, 3, 4, 5]
LinkedList after insertion and deletion: [0, 1, 99, 3, 4, 5, 100]
Forward traversal: 0 1 99 3 4 5 100
Backward traversal: 100 5 4 3 99 1 0
```

解释

- **创建并填充 ArrayList**:
  - `ArrayList` 被填充了数字 `1` 到 `5`。
  - 在索引 `2` 处插入了数字 `99`。
- **转换为 LinkedList 并进行操作**:
  - 使用 `ArrayList` 创建了一个 `LinkedList`。
  - 在 `LinkedList` 的开头插入了 `0`，在末尾插入了 `100`，并删除了索引 `2` 处的元素。
- **遍历 LinkedList**:
  - 使用 `ListIterator` 前向遍历 `LinkedList`。
  - 使用 `ListIterator` 后向遍历 `LinkedList`。

备注

- 列表可以包含重复元素。例如，在这个例子中，整数 `1` 被存储了两次：在原始 `ArrayList` 和转换后的 `LinkedList` 中。

5.Comparator interface

`java.util.Comparator<T>` interface

在Java中，`Comparator` 接口用于定义自定义的比较规则，用于比较不实现 `Comparable` 接口的对象，或者按不同于 `Comparable` 接口的标准进行比较。

`Comparator` 接口

`Comparator` 接口是一个函数式接口（在Java 8之后），用于定义自定义的比较逻辑。它有两个主要方法：

- `compare(T o1, T o2)`: 比较两个对象，如果第一个对象小于、等于或大于第二个对象，分别返回负整数、零或正整数。
- `equals(Object obj)`: 这个方法来自于 `Object` 类，一般不需要重写，因为默认实现已经满足大多数需求。

示例：`GeometricObjectComparator`

下面是一个实现 `Comparator` 接口的示例，比较两个几何对象的面积：

```
import java.util.Comparator;

public class GeometricObjectComparator implements Comparator<GeometricObject>, java.io.Serializable {
    // 实现比较逻辑
    public int compare(GeometricObject o1, GeometricObject o2) {
        double area1 = o1.getArea();
        double area2 = o2.getArea();
        if (area1 < area2) {
            return -1;
        } else if (area1 == area2) {
            return 0;
        } else {
            return 1;
        }
    }
}
```

详细解释

1. **实现`Comparator`接口**：

```
public class GeometricObjectComparator implements Comparator<GeometricObject>, java.io.Serializable {
```

   - `implements Comparator<GeometricObject>`：表示这个类定义了一个用于比较`GeometricObject`对象的比较器。
   - `java.io.Serializable`：使这个比较器可以序列化，这在需要将比较器存储在文件或通过网络传输时很有用。

2. **实现`compare`方法**：

```
public int compare(GeometricObject o1, GeometricObject o2) {
    double area1 = o1.getArea();
    double area2 = o2.getArea();
    if (area1 < area2) {
        return -1;
    } else if (area1 == area2) {
        return 0;
    } else {
        return 1;
    }
}
```

   - 获取两个几何对象的面积：`double area1 = o1.getArea();` 和 `double area2 = o2.getArea();`
   - 比较这两个面积：
     - 如果`area1`小于`area2`，返回-1。
     - 如果`area1`等于`area2`，返回0。
     - 如果`area1`大于`area2`，返回1。

使用示例

假设我们有一个`GeometricObject`类和一个列表需要排序：

```
import java.util.*;

class GeometricObject {
    private double area;

    public GeometricObject(double area) {
        this.area = area;
    }

    public double getArea() {
        return area;
    }
}

public class Main {
    public static void main(String[] args) {
        List<GeometricObject> list = new ArrayList<>();
        list.add(new GeometricObject(5.0));
        list.add(new GeometricObject(3.0));
        list.add(new GeometricObject(4.0));

        Collections.sort(list, new GeometricObjectComparator());

        for (GeometricObject obj : list) {
            System.out.println(obj.getArea());
```

```
    }
  }
}
```

输出：

```
3.0
4.0
5.0
```

总结

通过实现 `Comparator` 接口，可以自定义对象的比较逻辑，使得排序和其他比较操作更灵活和多样化。这样就能根据特定需求（例如面积、名字、年龄等）对对象进行排序。

6. Static Methods for Lists and Collections

Java `Collections` 类的静态方法

`Collections` 类提供了一系列静态方法，用于操作和处理集合（例如列表）。下面是这些方法的详细介绍和示例：

1. 排序

升序排序

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

public class SortExample {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("red", "green", "blue");
        Collections.sort(list);
        System.out.println(list);  // 输出: [blue, green, red]
    }
}
```

降序排序

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

public class SortReverseExample {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("red", "green", "blue", "yellow");
        Collections.sort(list, Collections.reverseOrder());
        System.out.println(list);  // 输出: [yellow, red, green, blue]
    }
}
```

2. 二分查找

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

```
public class BinarySearchExample {
    public static void main(String[] args) {
        List<Integer> list1 = Arrays.asList(2, 4, 7, 10, 11, 45, 50, 59, 60, 66);
        System.out.println("(1) Index: " + Collections.binarySearch(list1, 7));  // 输出: 2
        System.out.println("(2) Index: " + Collections.binarySearch(list1, 9));  // 输出: -4

        List<String> list2 = Arrays.asList("blue", "green", "red");
        System.out.println("(3) Index: " + Collections.binarySearch(list2, "red"));  // 输出: 2
        System.out.println("(4) Index: " + Collections.binarySearch(list2, "cyan"));  // 输出: -2
    }
}
```

3. 反转

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

public class ReverseExample {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("yellow", "red", "green", "blue");
        Collections.reverse(list);
        System.out.println(list);  // 输出: [blue, green, red, yellow]
    }
}
```

4. 洗牌

随机洗牌

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

public class ShuffleExample {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("yellow", "red", "green", "blue");
        Collections.shuffle(list);
        System.out.println(list);
    }
}
```

使用指定的随机对象洗牌

```
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import java.util.Random;

public class SameShuffleExample {
    public static void main(String[] args) {
        List<String> list1 = Arrays.asList("yellow", "red", "green", "blue");
        List<String> list2 = Arrays.asList("Y", "R", "G", "B");
        Collections.shuffle(list1, new Random(20));
        Collections.shuffle(list2, new Random(20));
        System.out.println(list1);  // 输出: [blue, yellow, red, green]
```

```
        System.out.println(list2);  // 输出: [B, Y, R, G]
    }
}
```

5. 复制

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

public class CopyExample {
    public static void main(String[] args) {
        List<String> list1 = Arrays.asList("yellow", "red", "green", "blue");
        List<String> list2 = Arrays.asList("white", "black");
        Collections.copy(list1, list2);
        System.out.println(list1);  // 输出: [white, black, green, blue]
    }
}
```

注意：如果目标列表比源列表小，会抛出 `IndexOutOfBoundsException` 异常。

6. 创建不可变的列表

```
import java.util.Arrays;
import java.util.Collections;
import java.util.GregorianCalendar;
import java.util.List;

public class NCopiesExample {
    public static void main(String[] args) {
        List<GregorianCalendar> list1 = Collections.nCopies(3, new GregorianCalendar(2022, 0, 1));
        System.out.println(list1);  // 输出: [java.util.GregorianCalendar[time=...], java.util.GregorianCalendar[time=...], java.util.GregorianCalendar[time=...]]
    }
}
```

7. 填充

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

public class FillExample {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("red", "green", "blue");
        Collections.fill(list, "black");
        System.out.println(list);  // 输出: [black, black, black]
    }
}
```

8. 查找最大值和最小值

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

```
public class MaxMinExample {
    public static void main(String[] args) {
        List<String> collection = Arrays.asList("red", "green", "blue");
        System.out.println(Collections.max(collection));  // 输出: red
        System.out.println(Collections.min(collection));  // 输出: blue
    }
}
```

9. 判定两个集合是否不相交

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

public class DisjointExample {
    public static void main(String[] args) {
        List<String> collection1 = Arrays.asList("red", "cyan");
        List<String> collection2 = Arrays.asList("red", "blue");
        List<String> collection3 = Arrays.asList("pink", "tan");

        System.out.println(Collections.disjoint(collection1, collection2));  // 输出: false
        System.out.println(Collections.disjoint(collection1, collection3));  // 输出: true
    }
}
```

10. 查找元素出现的频率

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

public class FrequencyExample {
    public static void main(String[] args) {
        List<String> collection = Arrays.asList("red", "cyan", "red");
        System.out.println(Collections.frequency(collection, "red"));  // 输出: 2

        List<String> collectionWithNewStrings = Arrays.asList(new String("red"), "cyan", new String("red"), "red");
        System.out.println(Collections.frequency(collectionWithNewStrings, "red"));  // 输出: 3
    }
}
```

总结

这些方法为我们提供了对集合进行各种常见操作的便捷手段，使得处理和操作集合变得更加高效和容易。通过这些示例，您可以更好地理解并应用这些方法来处理您的集合数据。

7. stack and queue

Vector 和 Stack 类

在 Java 2 之前，Java 已经支持了一些集合类，其中包括 Vector 和 Stack 类。在 JCF 中，这些类被重新设计以符合框架的标准，但为了确保向后兼容，保留了一些旧方法。

Vector 类

- **定义**：Vector 类类似于 ArrayList 类，区别在于 Vector 的方法是同步的（synchronized）。

- **同步方法**：Vector 的同步方法可以防止在多线程环境下数据被破坏。当多个线程同时访问和修改 Vector 时，同步方法可以确保线程安全。
- **效率**：对于不需要同步的应用程序，ArrayList 比 Vector 更高效。
- **旧方法**：`addElement(Object element)` 方法与 `add(Object element)` 方法相同，区别在于前者是同步的。

示例代码：

```
import java.util.Vector;

public class VectorExample {
    public static void main(String[] args) {
        Vector<String> vector = new Vector<>();
        vector.addElement("Java");
        vector.addElement("Python");
        vector.addElement("C++");

        for (String element : vector) {
            System.out.println(element);
        }
    }
}
```

Stack 类

- **定义**：Stack 类表示一个后进先出 (LIFO) 的对象堆栈。
- **操作**：Stack 类的操作包括 `push(o:E)` 插入元素、`peek()` 查看顶端元素、`pop()` 移除并返回顶端元素。
- **实现**：Stack 类是 Vector 类的子类。
- **旧方法**：`empty()` 方法与 `isEmpty()` 方法相同。

示例代码：

```
import java.util.Stack;

public class StackExample {
    public static void main(String[] args) {
        Stack<String> stack = new Stack<>();
        stack.push("Java");
        stack.push("Python");
        stack.push("C++");

        while (!stack.empty()) {
            System.out.println(stack.pop());
        }
    }
}
```

8.Queue 和 PriorityQueue

Queue 接口

- **定义**：Queue 是一个先进先出 (FIFO) 的数据结构。元素添加到队列的末尾，从队列的头部移除。
- **方法**：
  - `offer(o:E)`：向队列添加一个元素。
  - `poll()`：移除并返回队列头部的元素，队列为空时返回 `null`。
  - `remove()`：移除并返回队列头部的元素，队列为空时抛出异常。

- `peek()`：返回队列头部的元素但不移除，队列为空时返回 `null`。
- `element()`：返回队列头部的元素但不移除，队列为空时抛出异常。

PriorityQueue

- **定义**：优先级队列根据元素的自然顺序或自定义的比较器对元素进行排序。
- **方法**：
  - 默认情况下，值最小的元素有最高优先级。
  - 可以通过构造函数 `PriorityQueue(initialCapacity, comparator)` 指定排序方式。

示例代码：

```
import java.util.PriorityQueue;
import java.util.Collections;

public class PriorityQueueExample {
    public static void main(String[] args) {
        PriorityQueue<String> queue = new PriorityQueue<>();
        queue.offer("Oklahoma");
        queue.offer("Indiana");
        queue.offer("Georgia");
        queue.offer("Texas");

        System.out.println("Priority queue using Comparable:");
        while (!queue.isEmpty()) {
            System.out.print(queue.poll

```java
() + " ");
        }

        PriorityQueue<String> reversedQueue = new PriorityQueue<>(4, Collections.reverseOrder());
        reversedQueue.offer("Oklahoma");
        reversedQueue.offer("Indiana");
        reversedQueue.offer("Georgia");
        reversedQueue.offer("Texas");

        System.out.println("\\nPriority queue using Comparator:");
        while (!reversedQueue.isEmpty()) {
            System.out.print(reversedQueue.poll() + " ");
        }
    }
}
```

使用 LinkedList 实现队列

LinkedList 类实现了 Deque 接口，Deque 接口继承自 Queue 接口。

- **双端队列 (Deque)**：Deque 支持在两端插入和移除元素。方法包括 `addFirst(e)`、`removeFirst()`、`addLast(e)`、`removeLast()`、`getFirst()` 和 `getLast()`。

示例代码：

```
import java.util.LinkedList;
import java.util.Queue;

public class LinkedListQueueExample {
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();
```
```

```
        queue.offer("Oklahoma");
        queue.offer("Indiana");
        queue.offer("Georgia");
        queue.offer("Texas");

        while (!queue.isEmpty()) {
          System.out.print(queue.remove() + " ");
        }
    }
}
```

表达式求值的案例研究

使用堆栈求值表达式，特别是中缀表达式。

表达式求值算法

1. **第一阶段**：从左到右扫描中缀表达式，提取操作数、操作符和括号，并计算表达式的值。
    - 如果是操作数，将其推入操作数堆栈。
    - 如果是操作符，根据优先级处理操作符堆栈中的操作符，然后将当前操作符推入操作符堆栈。
    - 如果是左括号，将其推入操作符堆栈。
    - 如果是右括号，处理操作符堆栈中的操作符直到遇到左括号。
2. **第二阶段**：处理操作符堆栈中剩余的操作符，直到堆栈为空。

示例代码：

```
import java.util.Stack;

public class EvaluateExpression {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println("Usage: java EvaluateExpression \\"expression\\"");
            System.exit(1);
        }

        try {
            System.out.println(evaluateExpression(args[0]));
        } catch (Exception ex) {
            System.out.println("Wrong expression: " + args[0]);
        }
    }

    public static int evaluateExpression(String expression) {
        Stack<Integer> operandStack = new Stack<>();
        Stack<Character> operatorStack = new Stack<>();

        expression = insertBlanks(expression);
        String[] tokens = expression.split(" ");

        for (String token : tokens) {
            if (token.length() == 0) continue;
            else if (token.charAt(0) == '+' || token.charAt(0) == '-') {
                while (!operatorStack.isEmpty() &&
                        (operatorStack.peek() == '+' || operatorStack.peek() == '-' ||
                         operatorStack.peek() == '*' || operatorStack.peek() == '/')) {
                    processAnOperator(operandStack, operatorStack);
                }
```

```
            operatorStack.push(token.charAt(0));
        } else if (token.charAt(0) == '*' || token.charAt(0) == '/') {
            while (!operatorStack.isEmpty() &&
                    (operatorStack.peek() == '*' || operatorStack.peek() == '/')) {
                processAnOperator(operandStack, operatorStack);
            }
            operatorStack.push(token.charAt(0));
        } else if (token.trim().charAt(0) == '(') {
            operatorStack.push('(');
        } else if (token.trim().charAt(0) == ')') {
            while (operatorStack.peek() != '(') {
                processAnOperator(operandStack, operatorStack);
            }
            operatorStack.pop();
        } else {
            operandStack.push(Integer.parseInt(token));
        }
    }

    while (!operatorStack.isEmpty()) {
        processAnOperator(operandStack, operatorStack);
    }

    return operandStack.pop();
}

public static void processAnOperator(Stack<Integer> operandStack, Stack<Character> operatorStack) {
    char op = operatorStack.pop();
    int op1 = operandStack.pop();
    int op2 = operandStack.pop();
    switch (op) {
        case '+': operandStack.push(op2 + op1); break;
        case '-': operandStack.push(op2 - op1); break;
        case '*': operandStack.push(op2 * op1); break;
        case '/': operandStack.push(op2 / op1); break;
    }
}

public static String insertBlanks(String s) {
    StringBuilder result = new StringBuilder();
    for (int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);
        if (c == '(' || c == ')' || c == '+' || c == '-' || c == '*' || c == '/') {
            result.append(" ").append(c).append(" ");
        } else {
            result.append(c);
        }
    }
    return result.toString();
  }
}
```

通过这些示例和详细的解释，您应该可以更好地理解 Java 中的 Vector、Stack、Queue 和 PriorityQueue 类的使用和特点，以及如何使用堆栈进行表达式求值。

Lecture8 - Time complexity

1.GCD

1. **Greatest Common Divisors (GCD) Using Euclid’s Algorithm**:
   - Euclid's算法是一种高效地计算最大公约数的方法。通过反复应用余数运算，直到余数为0为止，最后的除数即为最大公约数。
   - `gcd(int m, int n)`方法使用递归实现了Euclid's算法。这种实现的时间复杂度是O(log n)。
2. **Efficient Algorithms for Finding Prime Numbers**:
   - 素数是只能被1和自身整除的正整数。素数的发现和研究一直是数学和计算机领域的重要课题之一。
   - 文中介绍了Brute-force和Sieve of Eratosthenes两种方法来找出小于给定数n的所有素数。
   - Brute-force方法通过逐个检查每个数的因子来判断其是否为素数。时间复杂度为O(n√n)。
   - Sieve of Eratosthenes算法通过逐步排除非素数来找出素数，其时间复杂度为O(n log n)。
3. **Computational Geometry: Finding a Convex Hull**:
   - 凸包是包围给定点集的最小凸多边形。凸包在计算机图形学和几何算法中有广泛应用。
   - 文中介绍了Gift-Wrapping算法（也称为Jarvis March）和Graham Scan算法两种寻找凸包的方法。
   - Gift-Wrapping算法的时间复杂度为O(hn)，其中h为凸包的大小，最坏情况下h为n，因此复杂度为O(n^2)。
   - Graham Scan算法的时间复杂度为O(n log n)。

现在让我们结合代码来进一步理解这些算法的实现：

```
public class Main {
    public static void main(String[] args) {
        // 使用Euclid's算法计算最大公约数
        int gcdResult = gcd(20, 15);
        System.out.println("GCD: " + gcdResult);

        // 使用Sieve of Eratosthenes算法找出小于等于20的所有素数
        findPrimes(20);

        // 使用Gift-Wrapping算法找出给定点集的凸包
        Point[] points = {new Point(0, 3), new Point(2, 2), new Point(1, 1), new Point(2, 1),
                          new Point(3, 0), new Point(0, 0), new Point(3, 3)};
        convexHull(points);
    }

    // Euclid's算法实现
    public static int gcd(int m, int n) {
        if (m % n == 0)
            return n;
        else
            return gcd(n, m % n);
    }

    // 使用Sieve of Eratosthenes算法找出小于等于n的所有素数
    public static void findPrimes(int n) {
        boolean[] primes = new boolean[n + 1];
        // 初始化素数数组
        for (int i = 2; i <= n; i++) {
            primes[i] = true;
        }

        // 找出素数
        for (int i = 2; i <= Math.sqrt(n); i++) {
            if (primes[i]) {
                for (int j = i * i; j <= n; j += i) {
```

```
                primes[j] = false;
            }
        }
    }

    // 输出素数
    System.out.println("Prime numbers less than or equal to " + n + ":");
    for (int i = 2; i <= n; i++) {
        if (primes[i]) {
            System.out.print(i + " ");
        }
    }
    System.out.println();
}

// Gift-Wrapping算法实现
public static void convexHull(Point[] points) {
    // TODO: 实现Gift-Wrapping算法
    }
}

// 定义Point类表示二维平面上的点
class Point {
    int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

这些代码片段展示了如何实现Euclid's算法，Sieve of Eratosthenes算法以及开始Gift-Wrapping算法。通过继续填充`convexHull`方法，你可以完成Gift-Wrapping算法的实现。

2.time complexity

线性算法 (Linear Algorithm)

定义: 具有 \(O(n)\) 时间复杂度的算法称为线性算法。

Definition: An algorithm with \(O(n)\) time complexity is called a linear algorithm.

增长速度: 线性算法的运行时间与输入规模成正比。

Growth Rate: The running time of a linear algorithm is directly proportional to the size of the input.

输入规模的影响: 如果输入规模加倍，算法的时间也会加倍。

Impact of Input Size: If you double the input size, the time for the algorithm will also double.

示例: 线性搜索算法在最坏情况下具有 \(O(n)\) 时间复杂度。

Example: The linear search algorithm has a time complexity of \(O(n)\) in the worst case.

```
public static int linearSearch(int[] list, int key) {
  for (int i = 0; i < list.length; i++) {
    if (list[i] == key) {
      return i;
    }
```

```
  }
  return -1;
}
```

常数算法 (Constant Algorithm)

定义: 具有 \(O(1)\) 时间复杂度的算法称为常数算法。

Definition: An algorithm with \(O(1)\) time complexity is called a constant algorithm.

增长速度: 常数算法的运行时间与输入规模无关。

Growth Rate: The running time of a constant algorithm is independent of the size of the input.

输入规模的影响: 无论输入规模多大，算法的时间都保持不变。

Impact of Input Size: The time for the algorithm remains constant regardless of the input size.

示例: 数组中获取一个元素的操作具有 \(O(1)\) 时间复杂度。

Example: Retrieving an element from an array has a time complexity of \(O(1)\).

```
public static int getElement(int[] list, int index) {
  return list[index];
}
```

指数算法 (Exponential Algorithm)

定义: 具有 \(O(2^n)\) 时间复杂度的算法称为指数算法。

Definition: An algorithm with \(O(2^n)\) time complexity is called an exponential algorithm.

增长速度: 指数算法的运行时间随着输入规模呈指数级增长。

Growth Rate: The running time of an exponential algorithm grows exponentially with the size of the input.

输入规模的影响: 如果输入规模加倍，算法的时间将会成倍增加。

Impact of Input Size: If you double the input size, the time for the algorithm increases exponentially.

示例: 计算斐波那契数列的递归算法在最坏情况下具有 \(O(2^n)\) 时间复杂度。

Example: The recursive algorithm for computing Fibonacci numbers has a time complexity of \(O(2^n)\) in the worst case.

```
public static int fibonacci(int n) {
  if (n <= 1) {
    return n;
  }
  return fibonacci(n - 1) + fibonacci(n - 2);
}
```

二次算法（quadratic algorithm）

定义: 具有 \(O(n^2)\) 时间复杂度的算法称为二次算法。

Definition: An algorithm with \(O(n^2)\) time complexity is called a quadratic algorithm.

增长速度: 二次算法随着问题规模的增加增长迅速。

Growth Rate: The quadratic algorithm grows quickly as the problem size increases.

输入规模的影响: 如果输入规模加倍，算法的时间会增加四倍。

Impact of Input Size: If you double the input size, the time for the algorithm is quadrupled.

嵌套循环: 含有嵌套循环的算法通常是二次算法。

Nested Loops: Algorithms with nested loops are often quadratic.

嵌套循环示例:

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    // 执行一些操作
  }
}
```

Explanation: 内层循环在每次外层循环中执行 \(n\) 次操作，总体时间复杂度为 \(O(n^2)\)。

Explanation: The inner loop executes \(n\) times for each iteration of the outer loop, resulting in an overall time complexity of \(O(n^2)\).

选择排序算法: 选择排序是一个具有二次时间复杂度的经典例子。

Selection Sort Algorithm: Selection sort is a classic example of an algorithm with quadratic time complexity.

```
public static void selectionSort(double[] list) {
  for (int i = 0; i < list.length; i++) {
    double currentMin = list[i];
    int currentMinIndex = i;
    for (int j = i + 1; j < list.length; j++) {
      if (currentMin > list[j]) {
        currentMin = list[j];
        currentMinIndex = j;
      }
    }
    if (currentMinIndex != i) {
      list[currentMinIndex] = list[i];
      list[i] = currentMin;
    }
  }
}
```

分析: 对于大小为 \(n\) 的数组，外层循环运行 \(n\) 次，内层循环在最坏情况下运行 \(n-1\) 次，总体时间复杂度为 \(O(n^2)\)。

Analysis: For an array of size \(n\), the outer loop runs \(n\) times, and the inner loop runs \(n-1\) times in the worst case, resulting in an overall time complexity of \(O(n^2)\).

对数算法 (Logarithmic Algorithm)

定义: 具有 \(O(\log n)\) 时间复杂度的算法称为对数算法。

Definition: An algorithm with \(O(\log n)\) time complexity is called a logarithmic algorithm.

增长速度: 对数算法的运行时间随着输入规模的对数增长。

Growth Rate: The running time of a logarithmic algorithm grows logarithmically with the size of the input.

输入规模的影响: 如果输入规模加倍，算法的时间仅增加一个常数。

Impact of Input Size: If you double the input size, the time for the algorithm increases by a constant amount.

示例: 二分搜索算法具有 \(O(\log n)\) 时间复杂度。

Example: The binary search algorithm has a time complexity of \(O(\log n)\).

```
public static int binarySearch(int[] list, int key) {
  int low = 0;
  int high = list.length - 1;
  while (low <= high) {
    int mid = (low + high) / 2;
    if (key < list[mid]) {
      high = mid - 1;
    } else if (key == list[mid]) {
      return mid;
    } else {
      low = mid + 1;
    }
  }
  return -1;
}
```

通过这些定义和示例，可以更好地理解不同时间复杂度的算法及其特征。

Lecture9 - Sorting

1.Bubble Sort

冒泡排序 (Bubble Sort)

冒泡排序是一种简单的排序算法，它通过重复遍历列表、比较相邻元素并在它们的顺序错误时进行交换来排序列表。

主要步骤:

1. 遍历列表，比较每对相邻元素，如果前一个元素大于后一个元素，则交换它们。
2. 对列表进行多次遍历，每次遍历后，未排序部分的最大元素会“冒泡”到列表的末端。
3. 重复上述过程，直到整个列表有序。

代码示例:

```
for (int k = 1; k < list.length; k++) {
    // 执行第 k 次遍历
    for (int i = 0; i < list.length - k; i++) {
        if (list[i] > list[i + 1]) {
            // 交换 list[i] 和 list[i + 1]
            int temp = list[i];
            list[i] = list[i + 1];
            list[i + 1] = temp;
        }
    }
}
```

冒泡排序过程

- 第一次遍历后，最大的元素移动到列表的末尾。
- 第二次遍历后，次大的元素移动到倒数第二的位置。
- 重复这个过程，直到列表排序完成。

冒泡排序之所以得名，是因为较大的值会逐渐“冒泡”到列表的顶端（数组的末尾），而较小的值会逐渐“下沉”到列表的底部（数组的开头）。

冒泡排序示例

给定数组: `[9, 8, 5, 4, 2, 1]`

1. 第一次遍历:
   - 比较并交换9和8: `[8, 9, 5, 4, 2, 1]`
   - 比较并交换9和5: `[8, 5, 9, 4, 2, 1]`
   - 比较并交换9和4: `[8, 5, 4, 9, 2, 1]`
   - 比较并交换9和2: `[8, 5, 4, 2, 9, 1]`
   - 比较并交换9和1: `[8, 5, 4, 2, 1, 9]`
2. 第二次遍历:
   - 比较并交换8和5: `[5, 8, 4, 2, 1, 9]`
   - 比较并交换8和4: `[5, 4, 8, 2, 1, 9]`
   - 比较并交换8和2: `[5, 4, 2, 8, 1, 9]`
   - 比较并交换8和1: `[5, 4, 2, 1, 8, 9]`
3. 第三次遍历:
   - 比较并交换5和4: `[4, 5, 2, 1, 8, 9]`
   - 比较并交换5和2: `[4, 2, 5, 1, 8, 9]`
   - 比较并交换5和1: `[4, 2, 1, 5, 8, 9]`
4. 第四次遍历:
   - 比较并交换4和2: `[2, 4, 1, 5, 8, 9]`
   - 比较并交换4和1: `[2, 1, 4, 5, 8, 9]`
5. 第五次遍历:
   - 比较并交换2和1: `[1, 2, 4, 5, 8, 9]`

此时数组已排序完成。

冒泡排序优化

如果在某次遍历中没有发生任何交换，则说明列表已经排序完毕，可以提前结束遍历过程。

优化后的代码示例：

```
public static void bubbleSort(int[] list) {
    boolean needNextPass = true;
    for (int k = 1; k < list.length && needNextPass; k++) {
        needNextPass = false;
        for (int i = 0; i < list.length - k; i++) {
            if (list[i] > list[i + 1]) {
                int temp = list[i];
                list[i] = list[i + 1];
                list[i + 1] = temp;
                needNextPass = true;
            }
        }
    }
}
```

冒泡排序分析

- **最佳情况**: 如果数组已经排序完毕，仅需一次遍历即可发现无需交换。最佳情况时间复杂度为 $O(n)$。

- **最坏情况**: 每次遍历都需要交换，最坏情况时间复杂度为 \(O(n^2) \)。

冒泡排序示例代码

```
public class BubbleSort {
    public static void bubbleSort(int[] list) {
        boolean needNextPass = true;
        for (int k = 1; k < list.length && needNextPass; k++) {
            needNextPass = false;
            for (int i = 0; i < list.length - k; i++) {
                if (list[i] > list[i + 1]) {
                    int temp = list[i];
                    list[i] = list[i + 1];
                    list[i + 1] = temp;
                    needNextPass = true;
                }
            }
        }
    }

    public static void main(String[] args) {
        int size = 100000;
        int[] a = new int[size];
        randomInitiate(a);
        long startTime = System.currentTimeMillis();
        bubbleSort(a);
        long endTime = System.currentTimeMillis();
        System.out.println((endTime - startTime) + "ms");
    }

    private static void randomInitiate(int[] a) {
        for (int i = 0; i < a.length; i++)
            a[i] = (int) (Math.random() * a.length);
    }
}
```

2. Merge Sort

归并排序 (Merge Sort)

归并排序是一种分治算法，由约翰·冯·诺伊曼（John von Neumann）在1945年发明。该算法通过递归地将未排序的列表分成较小的子列表，再将这些子列表合并成排序好的列表来实现排序。

算法步骤：

1. **分割**：将未排序的列表分成 `n` 个子列表，每个子列表包含一个元素（一个元素的列表被认为是排序好的）。
2. **合并**：重复合并排序好的子列表，直到只剩下一个子列表，这个子列表就是排序好的列表。

归并排序示例：

1. 初始列表：[2, 9, 5, 4, 8, 1, 6, 7]

   1.1. 分割步骤：

   ```
   [2, 9, 5, 4] 和 [8, 1, 6, 7]
   [2, 9] 和 [5, 4] 以及 [8, 1] 和 [6, 7]
   [2] 和 [9] 以及 [5] 和 [4] 以及 [8] 和 [1] 以及 [6] 和 [7]
   ```

   1.2. 合并步骤：

```
[2, 9] 和 [4, 5] 以及 [1, 8] 和 [6, 7]
[2, 4, 5, 9] 和 [1, 6, 7, 8]
[1, 2, 4, 5, 6, 7, 8, 9]
```

如何合并两个排序好的列表：

1. 比较两个列表的当前元素，将较小的元素添加到临时列表中。
2. 如果一个列表中的所有元素都已经添加完毕，则将另一个列表中的剩余元素依次添加到临时列表中。

合并两个排序好的列表的代码：

```
public static void merge(int[] list1, int[] list2, int[] temp) {
    int current1 = 0; // list1 的当前索引
    int current2 = 0; // list2 的当前索引
    int current3 = 0; // temp 的当前索引

    while (current1 < list1.length && current2 < list2.length) {
      if (list1[current1] < list2[current2])
        temp[current3++] = list1[current1++];
      else
        temp[current3++] = list2[current2++];
    }

    while (current1 < list1.length)
      temp[current3++] = list1[current1++];

    while (current2 < list2.length)
      temp[current3++] = list2[current2++];
}
```

归并排序的完整实现：

```
public class MergeSort {
    public static void mergeSort(int[] list) {
      if (list.length > 1) {
        // 归并排序前半部分
        int[] firstHalf = new int[list.length / 2];
        System.arraycopy(list, 0, firstHalf, 0, list.length / 2);
        mergeSort(firstHalf);

        // 归并排序后半部分
        int secondHalfLength = list.length - list.length / 2;
        int[] secondHalf = new int[secondHalfLength];
        System.arraycopy(list, list.length / 2, secondHalf, 0, secondHalfLength);
        mergeSort(secondHalf);

        // 将排序好的前半部分和后半部分合并
        merge(firstHalf, secondHalf, list);
    }
}

public static void merge(int[] list1, int[] list2, int[] temp) {
    int current1 = 0; // list1 的当前索引
    int current2 = 0; // list2 的当前索引
    int current3 = 0; // temp 的当前索引
```

```
    while (current1 < list1.length && current2 < list2.length) {
        if (list1[current1] < list2[current2])
            temp[current3++] = list1[current1++];
        else
            temp[current3++] = list2[current2++];
    }

    while (current1 < list1.length)
        temp[current3++] = list1[current1++];

    while (current2 < list2.length)
        temp[current3++] = list2[current2++];
}

public static void main(String[] args) {
    int[] list = {14, 12, 2, 3, 2, -2, 1, 3, 6, 5};
    mergeSort(list);
    for (int x : list)
        System.out.print(x + " ");
    // 输出: -2 1 2 2 3 3 5 6 12 14
  }
}
```

归并排序的时间复杂度分析

设 `T(n)` 为排序 `n` 个元素所需的时间。归并排序算法将数组分成两个子数组，递归地排序这些子数组，然后合并它们。

```
T(n) = 2T(n/2) + O(n)
```

- `2T(n/2)` 是排序两个子数组的时间。
- `O(n)` 是合并两个子数组所需的时间。

使用主定理，得到归并排序的时间复杂度为 $O(n \log n)$。

归并排序是一种高效的排序算法，尤其适合处理大规模的数据。它的主要优点是稳定且时间复杂度为 $O(n \log n)$。然而，归并排序的空间复杂度较高，需要额外的内存来存储临时数组。

3.Quick Sort

快速排序 (Quick Sort)

快速排序是由C.A.R. Hoare在1962年开发的一种分治算法。该算法通过选择一个枢轴元素(pivot)，将数组划分成两个部分，使得左边部分的所有元素都小于或等于枢轴，右边部分的所有元素都大于枢轴。然后递归地对这两个部分进行快速排序。

算法步骤：

1. **选择枢轴**：在数组中选择一个元素作为枢轴（通常选择第一个元素）。
2. **划分数组**：将数组分成两部分，左边部分所有元素小于或等于枢轴，右边部分所有元素大于枢轴。
3. **递归排序**：递归地对左边部分和右边部分进行快速排序。
4. **合并结果**：最后合并左边部分、枢轴和右边部分得到排序后的数组。

快速排序示例：

初始列表：[5, 2, 9, 3, 8, 4, 1, 6, 7]

1. 选择5作为枢轴：

   ```
   pivot = 5
   [5, 2, 9, 3, 8, 4, 1, 6, 7]
   ```

2. 划分数组：

   ```
   [2, 3, 4, 1] (<= pivot) 和 [9, 8, 6, 7] (> pivot)
   ```

3. 对子数组递归排序：

   ```
   [2, 3, 4, 1] 排序为 [1, 2, 3, 4]
   [9, 8, 6, 7] 排序为 [6, 7, 8, 9]
   ```

4. 合并结果：

   ```
   [1, 2, 3, 4] + [5] + [6, 7, 8, 9]
   最终结果：[1, 2, 3, 4, 5, 6, 7, 8, 9]
   ```

快速排序实现代码：

```
public class QuickSort {
    public static void quickSort(int[] list) {
        quickSort(list, 0, list.length - 1);
    }

    private static void quickSort(int[] list, int first, int last) {
        if (last > first) {
            int pivotIndex = partition(list, first, last);
            quickSort(list, first, pivotIndex - 1);
            quickSort(list, pivotIndex + 1, last);
        }
    }

    private static int partition(int[] list, int first, int last) {
        int pivot = list[first]; // 选择第一个元素作为枢轴
        int low = first + 1; // 向前搜索的索引
        int high = last; // 向后搜索的索引

        while (high > low) {
            // 从左边向右搜索
            while (low <= high && list[low] <= pivot)
                low++;
            // 从右边向左搜索
            while (low <= high && list[high] > pivot)
                high--;
            // 交换两个元素
            if (high > low) {
                int temp = list[high];
                list[high] = list[low];
                list[low] = temp;
            }
        }

        // 处理重复元素
        while (high > first && list[high] >= pivot)
            high--;

        // 将枢轴与list[high]交换
        if (pivot > list[high]) {
            list[first] = list[high];
            list[high] = pivot;
```

```
            return high;
        } else {
            return first;
        }
    }

    public static void main(String[] args) {
        int[] list = {14, 12, 2, 3, 2, -2, 1, 3, 6, 5};
        quickSort(list);
        for (int x : list)
            System.out.print(x + " ");
        // 输出: -2 1 2 2 3 3 5 6 12 14
    }
}
```

快速排序的时间复杂度分析

快速排序的时间复杂度主要取决于枢轴的选择：

- **最坏情况**：每次选择的枢轴将数组分成一个很小的部分和一个很大的部分（如已排序的数组），此时时间复杂度为 $O(n^2)$。
- **最佳情况**：每次选择的枢轴将数组均匀地分成两部分，此时时间复杂度为 $O(n \log n)$。
- **平均情况**：在大多数情况下，枢轴将数组分成大小接近的两部分，平均时间复杂度为 $O(n \log n)$。

总结

- **快速排序** 是一种高效的排序算法，通常在实践中比 **归并排序** 更快，因为它不需要额外的存储空间。

  Merge sort requires a temporary array for sorting two subarrays. Quick sort does not need additional array space. Thus, quick sort is **more space efficient** than merge sort.

4.Heap Sort

堆排序和二叉堆：概念与实现

二叉树

- **定义**：一种层次结构，每个节点最多有两个子节点。**Definition**: A hierarchical structure where each node has at most two children.
- **组成部分**：由一个根节点和两个分别称为左子树和右子树的二叉树组成。**Components**: Consists of a root node and two binary trees called the left subtree and right subtree.
- **路径长度**：路径中边的数量。**Path Length**: The number of edges in the path.
- **节点深度**：从根节点到该节点的路径长度。**Node Depth**: The length of the path from the root to that node.

完全二叉树

- **定义**：一个二叉树，除了最后一层，其他每一层都是满的，最后一层的节点从左到右依次排列。**Definition**: A binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

二叉堆

- **定义**：一个完全二叉树，其中每个节点的值都大于或等于其任何一个子节点的值。**Definition**: A complete binary tree where each node is greater than or equal to any of its children.
- **属性**：**Properties**:
  - 必须是完全二叉树。
    It must be a complete binary tree.
  - 每个节点的值都大于或等于其子节点的值（最大堆）。
    Each node's value is greater than or equal to its children's values (max heap).

堆排序算法

1. **构建堆**：将所有元素添加到堆中。**Heap Construction**: Add all elements to the heap.
2. **提取元素**：依次删除最大的元素以获得排序后的列表。**Element Extraction**: Successively remove the largest elements to obtain a sorted list.

堆的存储

- **数组表示**：可以用数组来存储堆。**Array Representation**: A heap can be stored in an array.
  - **父子关系**：**Parent-Child Relationship**:
    - 对于索引为 `i` 的节点，其左子节点在 `2i + 1` 位置，右子节点在 `2i + 2` 位置。
      For a node at index `i`, its left child is at `2i + 1` and its right child is at `2i + 2`.
    - 父节点在 `(i - 1) / 2` 位置。
      The parent node is at index `(i - 1) / 2`.

向堆中添加元素

1. 将元素添加到堆的末尾。
   Add the element to the end of the heap.
2. **上浮操作**：通过将新元素与其父节点比较并进行必要的交换来重建堆。**Sifting Up**: Rebuild the heap by comparing the new element with its parent and swapping if necessary.

删除根节点并重建堆

1. 用最后一个元素替换根节点。
   Replace the root with the last element.
2. **下沉操作**：通过将根节点与其子节点比较并与较大子节点交换来重建堆。**Sifting Down**: Rebuild the heap by comparing the root with its children and swapping with the larger child.

堆类实现（Java）

- **类定义**：**Class Definition**:

```
public class Heap<E extends Comparable> {
    private java.util.ArrayList<E> list = new java.util.ArrayList<E>();

    public Heap() {}

    public Heap(E[] objects) {
        for (E obj : objects) add(obj);
    }

    public void add(E newObject) {
        list.add(newObject);
        int currentIndex = list.size() - 1;
        while (currentIndex > 0) {
            int parentIndex = (currentIndex - 1) / 2;
            if (list.get(currentIndex).compareTo(list.get(parentIndex)) > 0) {
                E temp = list.get(currentIndex);
                list.set(currentIndex, list.get(parentIndex));
                list.set(parentIndex, temp);
            } else break;
            currentIndex = parentIndex;
        }
    }

    public E remove() {
        if (list.size() == 0) return null;
```

```
        E removedObject = list.get(0);
        list.set(0, list.get(list.size() - 1));
        list.remove(list.size() - 1);
        int currentIndex = 0;
        while (currentIndex < list.size()) {
            int leftChildIndex = 2 * currentIndex + 1;
            int rightChildIndex = 2 * currentIndex + 2;
            if (leftChildIndex >= list.size()) break;
            int maxIndex = leftChildIndex;
            if (rightChildIndex < list.size()) {
                if (list.get(maxIndex).compareTo(list.get(rightChildIndex)) < 0) {
                    maxIndex = rightChildIndex;
                }
            }
            if (list.get(currentIndex).compareTo(list.get(maxIndex)) < 0) {
                E temp = list.get(maxIndex);
                list.set(maxIndex, list.get(currentIndex));
                list.set(currentIndex, temp);
                currentIndex = maxIndex;
            } else break;
        }
        return removedObject;
    }

    public int getSize() {
        return list.size();
    }
}
```

堆排序实现（Java）

- **堆排序类：Heap Sort Class**:

```
public class HeapSort {
    public static <E extends Comparable> void heapSort(E[] list) {
        Heap<E> heap = new Heap<>();
        for (E e : list) heap.add(e);
        for (int i = list.length - 1; i >= 0; i--) list[i] = heap.remove();
    }

    public static void main(String[] args) {
        Integer[] list = {2, 3, 2, 5, 6, 1, -2, 3, 14, 12};
        heapSort(list);
        for (int i = 0; i < list.length; i++) System.out.print(list[i] + " ");
    }
}
```

堆排序的时间复杂度

- **构建堆**：O(n log n)，因为添加 n 个元素，每个需要 O(log n) 时间。**Heap Construction**: O(n log n), as adding n elements requires O(log n) time for each.
- **堆化操作**：O(n log n)，因为每次删除操作（下沉）需要 O(log n) 时间，重复 n 次。**Heap Operations**: O(n log n), as each remove operation (sifting down) takes O(log n) time, repeated n times.
- **总时间复杂度**：O(n log n)。**Total Time Complexity**: O(n log n).

与归并排序的比较

- **空间效率**：堆排序更节省空间，因为它不需要额外的数组空间。**Space Efficiency**: Heap sort is more space-efficient as it does not require additional array space.
- **时间复杂度**：**Time Complexity**:
  - **堆排序**：平均和最坏情况下都是 O(n log n)。**Heap Sort**: Both average and worst-case are O(n log n).
  - **归并排序**：一致的 O(n log n)，但需要额外的合并空间。**Merge Sort**: Consistently O(n log n), but requires extra merge space.

关键点

- **堆排序**：使用二叉堆的高效排序算法。**Heap Sort**: An efficient sorting algorithm using a binary heap.
- **二叉堆**：保持堆属性的完全二叉树。**Binary Heap**: A complete binary tree maintaining the heap property.
- **应用**：在实现优先队列和高效排序中非常有用。**Applications**: Very useful in implementing priority queues and efficient sorting.

5.Insert Sorting

插入排序算法：从概念到实现

插入排序是一种简单直观的排序算法，它通过逐步构建有序序列来完成排序任务。以下是详细的步骤和代码实现：

插入排序算法步骤

1. **初始状态**：只有第一个元素视为已排序，其余元素视为未排序。

   Initial State: Only the first element is considered sorted, the rest are unsorted.

2. **插入操作**：逐个元素从未排序部分取出，插入到已排序部分的正确位置，直到整个列表排序完毕。

   Insertion Operation: Each element from the unsorted part is taken and inserted into its correct position in the sorted part, until the whole list is sorted.

例子：逐步演示插入排序

给定无序列表 `myList = {2, 9, 5, 4, 8, 1, 6}`：

- **初始已排序子列表**：包含第一个元素 `[2]`。

  Initial sorted sublist: Contains the first element `[2]`.

1. **插入9**：
   - 比较9和2，由于9大于2，直接插入到已排序子列表的末尾。
   - 排序后的子列表为 `[2, 9]`。
2. **插入5**：
   - 比较5和9，5小于9，将9后移一位。
   - 比较5和2，5大于2，将5插入到2和9之间。
   - 排序后的子列表为 `[2, 5, 9]`。
3. **插入4**：
   - 比较4和9，4小于9，将9后移一位。
   - 比较4和5，4小于5，将5后移一位。
   - 比较4和2，4大于2，将4插入到2和5之间。
   - 排序后的子列表为 `[2, 4, 5, 9]`。
4. **插入8**：
   - 比较8和9，8小于9，将9后移一位。
   - 比较8和5，8大于5，将8插入到5和9之间。

- 排序后的子列表为 `[2, 4, 5, 8, 9]`。

5. **插入1**：
   - 依次比较1和9、8、5、4、2，1都小于它们，将这些元素依次后移一位。
   - 将1插入到列表开头。
   - 排序后的子列表为 `[1, 2, 4, 5, 8, 9]`。
6. **插入6**：
   - 比较6和9，6小于9，将9后移一位。
   - 比较6和8，6小于8，将8后移一位。
   - 比较6和5，6大于5，将6插入到5和8之间。
   - 排序后的列表为 `[1, 2, 4, 5, 6, 8, 9]`。

插入排序Java实现

```
public class InsertionSort {
    public static void main(String[] args) {
        int[] myList = {2, 9, 5, 4, 8, 1, 6};  // 无序数组
        insertionSort(myList);  // 调用插入排序算法
        for (int i : myList) {  // 输出排序后的数组
            System.out.print(i + " ");
        }
    }

    public static void insertionSort(int[] list) {
        for (int i = 1; i < list.length; i++) {  // 从第二个元素开始遍历
            int currentElement = list[i];  // 当前需要插入的元素
            int k;
            for (k = i - 1; k >= 0 && list[k] > currentElement; k--) {  // 找到当前元素的位置
                list[k + 1] = list[k];  // 向右移动元素
            }
            list[k + 1] = currentElement;  // 插入当前元素
        }
    }
}
```

插入排序算法分析

- **时间复杂度**：
  - 最坏情况：O(n^2)（数组完全逆序）。
  - 最好情况：O(n)（数组已经排序）。
  - 平均情况：O(n^2)。
- **空间复杂度**：O(1)，只需要常量级的额外空间。
- **稳定性**：插入排序是稳定的排序算法。

总结

插入排序通过将未排序的元素逐个插入到已排序部分，逐步构建有序序列。虽然在最坏情况下其时间复杂度较高，但在数据接近有序的情况下，插入排序的性能相对较好，并且它实现简单且无需额外的存储空间。

Lecture10 - Graph

1.Modeling Real-World Problems Using Graphs

用图模型化和解决实际问题

Graphs are useful in modeling and solving real-world problems.
图在建模和解决现实世界中的各种问题时非常有用。

Examples:

示例:

For example, finding the least number of flights between two cities is a shortest path problem in a graph.
例如，寻找两座城市之间的最少航班数是一个图中的最短路径问题。

Modeling Problems Using Graphs

用图模型化问题

Many practical problems can be represented by graphs because graphs are used to represent:
许多实际问题可以用图来表示，因为图用于表示：

- Travel routes (airline scheduling), optimal mail/package delivery, supply chain implementation.
- 旅行路线（如航空公司调度）、最优邮件/包裹配送、供应链实施。
- Networks of communication: routing is the selection of paths for traffic in a network.
- 通信网络：路由是为网络中的流量选择路径。
- Social media analysis: marketing (community detection), centrality measurement, information flow, maximizing influence, etc.
- 社交媒体分析：市场营销（社区检测）、中心性测量、信息流动、影响力最大化等。
- Computer chip design (placement of electronic components into an electrical network on a monolithic semiconductor).
- 计算机芯片设计（将电子元件放置在单片半导体上的电气网络中）。
- Search Engine Algorithms (e.g., PageRank algorithm).
- 搜索引擎算法（如PageRank算法）。

The development of algorithms to handle graphs is therefore of major interest in computer science.
因此，开发处理图的算法是计算机科学中的一个重要领域。

How it all started?

图论的起源

The study of graph problems is known as graph theory.
图论的研究起源于1736年，由莱昂哈德·欧拉（Leonhard Euler）提出，他通过图的术语解决了著名的柯尼斯堡七桥问题。

Leonhard Euler

莱昂哈德·欧拉

It was founded by Leonhard Euler in 1736, when he introduced graph terminology to solve the famous Seven Bridges of Königsberg problem.
柯尼斯堡七桥问题无解。

The Seven Bridges of Königsberg

柯尼斯堡七桥问题

The city of Königsberg, Prussia (now Kaliningrad, Russia), was divided by the Pregel River.
普鲁士的柯尼斯堡（现俄罗斯加里宁格勒）被普雷格尔河分隔。

There were two islands on the river.
河上有两座岛屿。

The city and islands were connected by seven bridges.
城市和岛屿之间由七座桥连接。

Euler replaced each land mass with a vertex (or a node), and each bridge with an edge:
欧拉将每个陆地区域替换为一个顶点，每座桥替换为一条边：

Euler's question: can one take a walk, cross each bridge exactly once, and return to the starting point?
欧拉的问题：是否存在一条路径可以经过每座桥一次且仅一次，并回到起点？

Euler proved that for such a path to exist, each vertex must have an even number of edges.
欧拉证明了若存在这样的路径，则每个顶点必须有偶数条边。

Therefore, the Seven Bridges of Königsberg problem has no solution!
因此，柯尼斯堡七桥问题无解！

Basic Graph Terminology

2.图的基本术语

A graph $G = (V, E)$, where V represents a set of vertices (or nodes) and E represents a set of edges (or links).
图 $G = (V, E)$，其中 V 代表顶点集合，E 代表边集合。

A graph may be undirected (i.e., if (x,y) is in E, then (y,x) is also in E) or directed.
图可以是无向图（即，如果 (x,y) 在 E 中，那么 (y,x) 也在 E 中）或有向图。

Adjacent Vertices

相邻顶点

Two vertices in a graph are said to be adjacent (or neighbors) if they are connected by an edge.
如果两个顶点由一条边连接，则它们是相邻顶点。

An edge in a graph that joins two vertices is said to be incident to both vertices.
连接两个顶点的边称为连接两个顶点的边。

For example, A and B are adjacent.
例如，A 和 B 是相邻顶点。

Degree

顶点的度

The degree of a vertex is the number of edges incident to it.
顶点的度是与其相连接的边的数目。

Complete Graph

完全图

In a complete graph, every two pairs of vertices are connected.
在完全图中，每对顶点之间都有一条边相连。

Incomplete Graph

非完全图

In an incomplete graph, not all pairs of vertices are connected.
在非完全图中，并不是所有顶点对都相连。

Unweighted and Weighted Graphs

无权图和有权图

An unweighted graph does not assign weights to edges, while a weighted graph does.
无权图的边没有权重，有权图的边有权重。

Parallel Edges and Loops

平行边和环

If two vertices are connected by two or more edges, these edges are called parallel edges.
如果两个顶点之间有多条边，这些边称为平行边。

A loop is an edge that links a vertex to itself.
一个环是连接顶点自身的一条边。

A simple graph is one that doesn't have any parallel edges or loops.
一个简单图没有任何平行边或环。

Connected Graph

连通图

A graph is connected if there exists a path between any two vertices in the graph.
如果图中任意两顶点之间存在路径，则该图为连通图。

Tree

树

A connected graph is a tree if it does not have cycles.
一个连通且无环的图是一棵树。

Cycles

环

A closed path is a path where all vertices have 2 edges incident to them.
一个闭合路径是所有顶点都有两条连接边的路径。

A cycle is a closed path that starts from a vertex and ends at the same vertex.
一个环是从一个顶点出发，经过若干条边回到该顶点的路径。

Subgraphs

子图

A subgraph of a graph G is a graph whose vertex set is a subset of that of G and whose edge set is a subset of that of G.
图 G 的子图是其顶点集合和边集合的子集构成的图。

Spanning Tree

生成树

A spanning tree of a graph G is a **connected subgraph** of G and the subgraph is a tree that **contains all vertices** in G.
图 G 的生成树是包含 G 所有顶点的连通子图。

3.Representing Graphs

图的表示方法

顶点的表示

- 可以使用字符串数组来表示顶点：

```
String[] vertices = {"Seattle", "San Francisco", "Los Angeles", "Denver", "Kansas City", "Chicago",...};
```

- 或者使用 `List<String>` 来动态添加顶点：

```
List<String> vertices;
vertices.add("Seattle");...
```

- 也可以定义一个 `City` 类来表示城市，并使用 `City` 数组：

```
public class City {
  private String cityName;
}
City[] vertices = {city0, city1, ... };
```

在所有这些表示方法中，顶点都可以用索引 0, 1, 2, ..., n-1 来标记，对于 n 个顶点的图。

边的表示方法

使用边数组表示边

- 可以使用一个二维数组来表示所有的边：

```
int[][] edges = {
  {0, 1}, {0, 3}, {0, 5}, // 从顶点 0 开始的边
  {1, 0}, {1, 2}, {1, 3}, // 从顶点 1 开始的边
  {2, 1}, {2, 3}, {2, 4}, {2, 10},
  {3, 0}, {3, 1}, {3, 2}, {3, 4}, {3, 5},
  {4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},
  {5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},
  {6, 5}, {6, 7},
  {7, 4}, {7, 5}, {7, 6}, {7, 8},
  {8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},
  {9, 8}, {9, 11},
  {10, 2}, {10, 4}, {10, 8}, {10, 11},
  {11, 8}, {11, 9}, {11, 10}
};
```

使用边对象表示边

- 定义一个 `Edge` 类来表示边，并使用 `ArrayList` 来存储边对象：

```
public class Edge {
  int u, v;
  public Edge(int u, int v) {
    this.u = u;
    this.v = v;
  }
}
List<Edge> list = new ArrayList();
list.add(new Edge(0, 1));
list.add(new Edge(0, 3));
```

使用邻接矩阵表示边

- 对于 n 个顶点的图，可以使用 n * n 的二维矩阵来表示边的存在：

```
int[][] adjacencyMatrix = {
  {0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0}, // Seattle
  {1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0}, // San Francisco
  {0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0}, // Los Angeles
  {1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0}, // Denver
  {0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0}, // Kansas City
  {1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0}, // Chicago
  {0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0}, // Boston
  {0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0}, // New York
  {0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1}, // Atlanta
```

```
  {0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1}, // Miami
  {0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1}, // Dallas
  {0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0}  // Houston
};
```

使用邻接表表示边

- 使用数组来存储每个顶点的邻接点列表：

```
List<Integer>[] neighbors = new List[12];
neighbors[0] = Arrays.asList(1, 3, 5); // Seattle
neighbors[1] = Arrays.asList(0, 2, 3); // San Francisco
// 继续为其他顶点添加邻接点...
```

使用邻接边表表示边

- 使用 `List<Edge>[]` 来存储每个顶点的邻接边列表：

```
List<Edge>[] neighbors = new List[12];
neighbors[0] = Arrays.asList(new Edge(0, 1), new Edge(0, 3), new Edge(0, 5)); // Seattle
neighbors[1] = Arrays.asList(new Edge(1, 0), new Edge(1, 2), new Edge(1, 3)); // San Francisco
// 继续为其他顶点添加邻接边...
```

图的建模

- `Graph` 接口定义了图的通用操作：

```
public interface Graph<V> {
  boolean addVertex(V vertex);
  boolean addEdge(int u, int v);
  AbstractGraph<V>.Tree dfs(int v);
  AbstractGraph<V>.Tree bfs(int v);
  int getSize();
  java.util.List<V> getVertices();
  V getVertex(int index);
  int getIndex(V v);
  java.util.List<Integer> getNeighbors(int index);
  int getDegree(int v);
  void printEdges();
  void clear();
}
```

- `AbstractGraph` 抽象类部分实现了 `Graph` 接口：

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

public abstract class AbstractGraph<V> implements Graph<V> {
  protected List<V> vertices = new ArrayList<>();
  protected List<List<Edge>> neighbors = new ArrayList<>();

  public static class Edge {
    public int u, v;
    public Edge(int u, int v) {
      this.u = u;
      this.v = v;
    }
    public boolean equals(Object o) {
```

```
    return u == ((Edge) o).u && v == ((Edge) o).v;
  }
}

@Override
public boolean addVertex(V vertex) {
  if (!vertices.contains(vertex)) {
    vertices.add(vertex);
    neighbors.add(new ArrayList<Edge>());
    return true;
  } else {
    return false;
  }
}

@Override
public boolean addEdge(int u, int v) {
  return addEdge(new Edge(u, v));
}

protected boolean addEdge(Edge e) {
  if (e.u < 0 || e.u > getSize() - 1)
    throw new IllegalArgumentException("No such index: " + e.u);
  if (e.v < 0 || e.v > getSize() - 1)
    throw new IllegalArgumentException("No such index: " + e.v);
  if (!neighbors.get(e.u).contains(e)) {
    neighbors.get(e.u).add(e);
    return true;
  } else {
    return false;
  }
}

@Override
public List<V> getVertices() {
  return vertices;
}

@Override
public V getVertex(int index) {
  return vertices.get(index);
}

@Override
public int getIndex(V v) {
  return vertices.indexOf(v);
}

@Override
public int getSize() {
  return vertices.size();
}

@Override
public List<Integer> getNeighbors(int index) {
  List<Integer> result = new ArrayList<>();
  for (Edge e : neighbors.get(index))
    result.add(e.v);
```

```
  return result;
}

@Override
public int getDegree(int u) {
  return neighbors.get(u).size();
}

@Override
public void printEdges() {
  for (int u = 0; u < neighbors.size(); u++) {
    System.out.print(getVertex(u) + " (" + u + "): ");
    for (Edge e : neighbors.get(u)) {
      System.out.print("(" + getVertex(e.u) + ", " + getVertex(e.v) + ") ");
    }
    System.out.println();
  }
}

@Override
public void clear() {
  vertices.clear();
  neighbors.clear();
}

@Override
public AbstractGraph<V>.Tree dfs(int v) {
  List<Integer> searchOrder = new ArrayList<>();
  int[] parent = new int[vertices.size()];
  for (int i = 0; i < parent.length; i++)
    parent[i] = -1;
  boolean[] isVisited = new boolean[vertices.size()];
  dfs(v, parent, searchOrder, isVisited);
  return new Tree(v, parent, searchOrder);
}

private void dfs(int u, int[] parent, List<Integer> searchOrder, boolean[] isVisited) {
  searchOrder.add(u);
  isVisited[u] = true;
  for (Edge e : neighbors.get(u)) {
    if (!isVisited[e.v]) {
      parent[e.v] = u;
      dfs(e.v, parent, searchOrder, isVisited);
    }
  }
}

@Override
public AbstractGraph<V>.Tree bfs(int v) {
  List<Integer> searchOrder = new ArrayList<>();
  int[] parent = new int[vertices.size()];
  for (int i = 0; i < parent.length; i++)
    parent[i] = -1;
  java.util.LinkedList<Integer> queue = new java.util.LinkedList<>();
  boolean[] isVisited = new boolean[vertices.size()];
  queue.offer(v);
  isVisited[v] = true;
  while (!queue.isEmpty()) {
```

```
    int u = queue.poll();
    searchOrder.add(u);
    for (Edge e : neighbors.get(u)) {
      if (!isVisited[e.v]) {
        queue.offer(e.v);
        parent[e.v] = u;
        isVisited[e.v] = true;
      }
    }
  }
  return new Tree(v, parent, searchOrder);
}

public class Tree {
  private int root;
  private int[] parent;
  private List<Integer> searchOrder;

public Tree(int root, int[] parent, List<Integer> searchOrder) {
  this.root = root;
  this.parent = parent;
  this.searchOrder = searchOrder;
}

public int getRoot() {
  return root;
}

public int getParent(int v) {
  return parent[v];
}

public List<Integer> getSearchOrder() {
  return searchOrder;
}

public int getNumberOfVerticesFound() {
  return searchOrder.size();
}

public List<V> getPath(int index) {
  ArrayList<V> path = new ArrayList<>();
  do {
    path.add(vertices.get(index));
    index = parent[index];
  } while (index != -1);
  return path;
}

public void printPath(int index) {
  List<V> path = getPath(index);
  System.out.print("A path from " + vertices.get(root) + " to " + vertices.get(index) + ": ");
  for (int i = path.size() - 1; i >= 0; i--)
    System.out.print(path.get(i) + " ");
}

public void printTree() {
  System.out.println("Root is: " + vertices.get(root));
```

```
      System.out.print("Edges: ");
      for (int i = 0; i < parent.length; i++) {
        if (parent[i] != -1) {
          System.out.print("(" + vertices.get(parent[i]) + ", " + vertices.get(i) + ") ");
        }
      }
      System.out.println();
    }
  }
}
```

- 使用 AbstractGraph 类- 具体的图类可以继承 `AbstractGraph` 类并实现抽象方法。

```
import java.util.List;

public class UnweightedGraph<V> extends AbstractGraph<V> {
  public UnweightedGraph() {
  }

  public UnweightedGraph(V[] vertices, int[][] edges) {
    for (int i = 0; i < vertices.length; i++)
      addVertex(vertices[i]);
    createAdjacencyLists(edges, vertices.length);
  }

  public UnweightedGraph(List<V> vertices, List<Edge> edges) {
    for (int i = 0; i < vertices.size(); i++)
      addVertex(vertices.get(i));
    createAdjacencyLists(edges, vertices.size());
  }

  public UnweightedGraph(List<Edge> edges, int numberOfVertices) {
    for (int i = 0; i < numberOfVertices; i++)
      addVertex((V)(new Integer(i))); // vertices is {0, 1, 2, . . . }
    createAdjacencyLists(edges, numberOfVertices);
  }

  public UnweightedGraph(int[][] edges, int numberOfVertices) {
    for (int i = 0; i < numberOfVertices; i++)
      addVertex((V)(new Integer(i))); // vertices is {0, 1, 2, . . . }
    createAdjacencyLists(edges, numberOfVertices);
  }

  private void createAdjacencyLists(int[][] edges, int numberOfVertices) {
    for (int i = 0; i < edges.length; i++) {
      addEdge(edges[i][0], edges[i][1]);
    }
  }

  private void createAdjacencyLists(List<Edge> edges, int numberOfVertices) {
    for (Edge edge : edges) {
      addEdge(edge.u, edge.v);
    }
  }
}
```

通过上述代码实现，可以建模和表示一个图，并实现常用的图算法。

Lecture11 - Tree

1.BST(不重复)

Binary Trees 二叉树

A binary tree is a hierarchical structure: it is either empty or consists of an element, called the root, and two distinct binary trees, called the left subtree and right subtree.

二叉树是一种层次结构：它要么为空，要么由一个元素（称为根节点）和两个不同的二叉树（称为左子树和右子树）组成。

Key Concepts 关键概念

- **Root (根节点):** The top element of the tree. 树的顶层元素。
- **Left Child (左子节点):** The root of the left subtree. 左子树的根节点。
- **Right Child (右子节点):** The root of the right subtree. 右子树的根节点。
- **Leaf (叶子节点):** A node without children. 没有子节点的节点。

Representing Binary Trees 表示二叉树

A binary tree can be represented using a set of linked nodes where each node contains an element value and two links named `left` and `right` that reference the left child and right child respectively.

可以使用一组链节点来表示二叉树，每个节点包含一个元素值和两个名为`left`和`right`的链接，分别引用左子节点和右子节点。

```
class TreeNode<E> {
    E element;
    TreeNode<E> left;
    TreeNode<E> right;

    public TreeNode(E o) {
        element = o;
    }
}
```

Representing Non-Binary Trees 表示非二叉树

Non-binary trees can be represented using a set of linked nodes with arrays or `ArrayList` to store the children.

可以使用一组链节点和数组或`ArrayList`来存储子节点，从而表示非二叉树。

```
class TreeNode<E> {
    E element;
    TreeNode<E>[] children; // Using array for children 使用数组存储子节点

    public TreeNode(E o) {
        element = o;
    }
}

// OR 或者

class TreeNode<E> {
    E element;
    ArrayList<TreeNode<E>> children; // Using ArrayList for children 使用ArrayList存储子节点
```

```
    public TreeNode(E o) {
        element = o;
    }
}
```

Binary Search Trees (BST) 二叉搜索树

A Binary Search Tree is a special type of binary tree that:

二叉搜索树是一种特殊的二叉树，它具有以下特点：

- Has no duplicate elements. 没有重复的元素。
- For every node in the tree, the value of any node in its left subtree is less than the value of the node. 树中每个节点，其左子树中任何节点的值都小于该节点的值。
- The value of any node in its right subtree is greater than the value of the node. 右子树中任何节点的值都大于该节点的值。

Inserting an Element into a BST 向二叉搜索树中插入元素

If the BST is empty, create a root node with the new element. Otherwise, insert the element into a leaf by:

如果BST为空，则使用新元素创建一个根节点。否则，将元素插入到叶子节点，通过以下步骤：

1. Locating the parent node for the new element node. 定位新元素节点的父节点。
2. Comparing the new element with the current node's element. 将新元素与当前节点的元素进行比较。
3. Recursively moving to the left or right child depending on the comparison. 根据比较结果递归地移动到左子节点或右子节点。
4. Inserting the new element as a left or right child of the parent node. 将新元素插入为父节点的左子节点或右子节点。

Implementation 实现

Here's how you can implement the `insert` method in Java:

以下是如何在Java中实现`insert`方法：

```
public class BST<E extends Comparable<E>> {
    protected TreeNode<E> root;

    // Method to insert an element into the BST 向BST插入元素的方法
    public boolean insert(E element) {
        if (root == null) { // If the tree is empty, create the root node 如果树为空，创建根节点
            root = new TreeNode<>(element);
        } else {
            // Locate the parent node 定位父节点
            TreeNode<E> current = root, parent = null;
            while (current != null) {
                if (element.compareTo(current.element) < 0) {
                    parent = current;
                    current = current.left;
                } else if (element.compareTo(current.element) > 0) {
                    parent = current;
                    current = current.right;
                } else {
                    return false; // Duplicate node not inserted 不插入重复节点
                }
            }
            // Create the new node and attach it to the parent node 创建新节点并将其附加到父节点
            if (element.compareTo(parent.element) < 0) {
                parent.left = new TreeNode<>(element);
            } else {
                parent.right = new TreeNode<>(element);
```

```
        }
      }
      return true; // Element inserted 元素插入成功
    }
  }
```

Tracing the Insertion of 101 into a BST 追踪将101插入BST的过程

Given a BST, let's trace the steps to insert 101:

给定一个BST，让我们追踪插入101的步骤：

1. Start with the root of the tree. 从树的根节点开始。
2. Compare 101 with the current node's element. 将101与当前节点的元素进行比较。
3. If 101 is greater, move to the right child. If 101 is less, move to the left child. 如果101更大，移动到右子节点。如果101更小，移动到左子节点。
4. Repeat until you find an appropriate leaf position for 101. 重复上述步骤，直到找到101的合适叶子位置。
5. Insert 101 as the right or left child of the determined parent node. 将101插入为确定的父节点的右子节点或左子节点。

For example, if the BST initially looks like this:

例如，如果BST最初看起来像这样：

```
     60
    /   \\
   55    100
  / \\   / \\
 45  57 67  107
```

Inserting 101: 插入101：

- Start at root (60). 101 > 60, move to the right child (100). 从根节点（60）开始。101 > 60，移动到右子节点（100）。
- Compare 101 with 100. 101 > 100, move to the right child (107). 将101与100进行比较。101 > 100，移动到右子节点（107）。
- Compare 101 with 107. 101 < 107, so insert 101 as the left child of 107. 将101与107进行比较。101 < 107，因此将101插入为107的左子节点。

The updated BST would be: 更新后的BST将是：

```
     60
    /   \\
   55    100
  / \\   / \\
 45  57 67  107
        /
       101
```

Complete Code for BST Insertion BST插入的完整代码

Here's the complete code for the BST class including the `TreeNode` class and `insert` method:

以下是包含`TreeNode`类和`insert`方法的BST类的完整代码：

```
public class BST<E extends Comparable<E>> {
    protected TreeNode<E> root;

    // Inner TreeNode class 内部TreeNode类
    static class TreeNode<E> {
        E element;
        TreeNode<E> left;
        TreeNode<E> right;
```

```
    public TreeNode(E o) {
        element = o;
    }
}

// Method to insert an element into the BST 向BST插入元素的方法
public boolean insert(E element) {
    if (root == null) { // If the tree is empty, create the root node 如果树为空，创建根节点
        root = new TreeNode<>(element);
    } else {
        // Locate the parent node 定位父节点
        TreeNode<E> current = root, parent = null;
        while (current != null) {
            if (element.compareTo(current.element) < 0) {
                parent = current;
                current = current.left;
            } else if (element.compareTo(current.element) > 0) {
                parent = current;
                current = current.right;
            } else {
                return false; // Duplicate node not inserted 不插入重复节点
            }
        }
        // Create the new node and attach it to the parent node 创建新节点并将其附加到父节点
        if (element.compareTo(parent.element) < 0) {
            parent.left = new TreeNode<>(element);
        } else {
            parent.right = new TreeNode<>(element);
        }
    }
    return true; // Element inserted 元素插入成功
}

// Main method for testing 主方法用于测试
public static void main(String[] args) {
    BST<Integer> bst = new BST<>();
    bst.insert(60);
    bst.insert(55);
    bst.insert(100);
        bst.insert(45);
        bst.insert(57);
        bst.insert(67);
        bst.insert(107);



 // Insert 101 and print the structure 插入101并打印结构
        bst.insert(101);

        // You can add a method to print the tree to verify the structure 您可以添加一个方法来打印树以验证结构
    }
}
```

You can add additional methods to print or traverse the tree to verify its structure after inserting elements.

您可以添加其他方法来打印或遍历树，以在插入元素后验证其结构。

二叉搜索树（BST）操作和树遍历

在二叉搜索树中搜索元素

BST中的搜索操作涉及根据二分搜索性质从根节点到所需节点的遍历。以下是其工作原理：

```
public boolean search(E element) {
    // 从根节点开始
    TreeNode<E> current = root;
    while (current != null) {
        if (element.compareTo(current.element) < 0) {
            current = current.left; // 向左移动
        } else if (element.compareTo(current.element) > 0) {
            current = current.right; // 向右移动
        } else {
            return true; // 元素已找到
        }
    }
    return false; // 元素不在树中
}
```

树的遍历

树的遍历是访问树中每个节点恰好一次的过程。有几种遍历方法：

1. **前序遍历**
    - 首先访问当前节点，然后递归访问左子树，最后递归访问右子树。
    - 示例：`60, 55, 45, 57, 100, 67, 107`
    - 应用：打印目录表。
2. **中序遍历**
    - 递归访问左子树，然后访问当前节点，最后递归访问右子树。
    - 示例：`45, 55, 57, 60, 67, 100, 107`
    - 应用：从树中读取算术表达式（BST的中序遍历即有序）。
3. **后序遍历**
    - 递归访问左子树，然后访问右子树，最后访问当前节点。
    - 示例：`45, 57, 55, 67, 107, 100, 60`
    - 应用：某些语言中表示表达式以进行高效解析。
4. **广度优先遍历（层次遍历）**
    - 逐层访问节点：先根节点，然后是根节点的所有子节点，以此类推。
    - 示例：`60, 55, 100, 45, 57, 67, 107`
    - 应用：图算法用于找到最短路径。
5. **深度优先遍历**
    - 分支地逐个访问节点。
    - 示例：对于二叉树，与前序遍历相同。

树接口和抽象树类

`Tree`接口定义了树的常见操作，`AbstractTree`类提供了部分实现：

```
public interface Tree<E> extends Iterable<E> {
    public boolean search(E e);
    public boolean insert(E e);
    public boolean delete(E e);
    public void preorder();
    public void inorder();
```

```
    public void postorder();
    public int getSize();
    public boolean isEmpty();
}

public abstract class AbstractTree<E> implements Tree<E> {
    @Override public void preorder() {}
    @Override public void inorder() {}
    @Override public void postorder() {}
    @Override public boolean isEmpty() { return getSize() == 0; }
}
```

BST类

`BST`类扩展了`AbstractTree`并实现了二叉搜索树的必要方法：

```
public class BST<E extends Comparable<E>> extends AbstractTree<E> {
    protected TreeNode<E> root;
    protected int size = 0;

    public static class TreeNode<E extends Comparable<E>> {
        protected E element;
        protected TreeNode<E> left;
        protected TreeNode<E> right;
        public TreeNode(E e) { element = e; }
    }

    public BST() {}
    public BST(E[] objects) {
        for (int i = 0; i < objects.length; i++) insert(objects[i]);
    }

    @Override public boolean search(E e) { /* 实现 */ }
    @Override public boolean insert(E e) { /* 实现 */ }
    @Override public boolean delete(E e) { /* 实现 */ }
    @Override public void preorder() { preorder(root); }
    @Override public void inorder() { inorder(root); }
    @Override public void postorder() { postorder(root); }
    @Override public int getSize() { return size; }

    // 其他方法和遍历实现...
}
```

在二叉搜索树中删除元素

删除过程涉及找到要删除的节点及其父节点，并处理两种主要情况：

1. **没有左子节点的节点**：
   - 将父节点与当前节点的右子节点连接。
2. **有左子节点的节点**：
   - 找到当前节点左子树中的最右节点，并用该节点替换当前节点。然后删除最右节点。

2. Huffman Tree

Huffman 编码是一种常用的数据压缩技术，能够有效地减少数据存储或传输所需的比特数。让我把你提供的信息梳理一下：

1. Huffman 编码原理：

- 每个字符用固定数量的比特表示，例如 ASCII 中每个字符占用 8 位。
- Huffman 编码根据字符出现的频率来分配比特，频率高的字符使用较短的编码，频率低的字符使用较长的编码。
- Huffman 编码通过构建一个频率排序的二叉树来实现，每个字符对应树的一个叶子节点，字符的编码由根节点到叶子节点的路径上的边值确定。

2. Huffman 树构建过程：

- 使用贪心算法构建 Huffman 树。
- 初始状态下，将每个字符看作一个独立的树，树的权重为字符在文本中出现的频率。
- 重复以下步骤，直到只剩下一个树：
  1. 选择权重最小的两棵树（使用最小堆实现的优先队列）。
  2. 将它们合并为一棵新的树，新树的权重为两棵子树的权重之和。

3. 示例程序功能：

- 输入一个文本。
- 统计文本中字符的频率。
- 构建 Huffman 树。
- 获取字符的 Huffman 编码。
- 对文本进行编码和解码。

以上是 Huffman 编码的核心原理和实现步骤。你可以使用示例程序来尝试编码和解码文本，以便更好地理解 Huffman 编码的工作原理。

Lecture12 - AVL and Hash

Why AVL Trees? 为什么选择AVL树？

- **Height-Dependent Operations**: The search, insertion, and deletion time for a binary search tree (BST) depend on the height of the tree. 搜索、插入和删除操作的时间取决于二叉搜索树的高度。
- **Worst Case Scenario**: In the worst case, the height of a BST can be O(n), leading to a worst-case time complexity of O(n). 在最坏情况下，BST的高度可以达到O(n)，导致最坏情况下的时间复杂度为O(n)。
- **Balanced Tree Advantage**: In a perfectly balanced tree, such as a complete binary tree, the height is log n, resulting in search, insertion, and deletion times of O(log n). 在一个完全平衡的树中，例如完全二叉树，其高度为log n，搜索、插入和删除操作的时间为O(log n)。
- **Balancing Trade-off**: Maintaining a perfectly balanced tree can be costly, so a compromise is to maintain a well-balanced tree where the heights of two subtrees for every node are approximately the same. 维护一个完全平衡的树代价很高，因此折衷方法是保持一个良好平衡的树，即每个节点的两个子树高度大致相同。

AVL Trees AVL树

- **Origins**: AVL trees are well-balanced binary search trees invented by Russian computer scientists Georgy Adelson-Velsky and Evgenii Landis in 1962. AVL树是由俄罗斯计算机科学家Georgy Adelson-Velsky和Evgenii Landis在1962年发明的良好平衡的二叉搜索树。
- **Balance Criterion**: In an AVL tree, the difference between the heights of two subtrees for every node is either 0 or 1. 在AVL树中，每个节点的两个子树高度差为0或1。
- **Height Efficiency**: The maximum height of an AVL tree is O(log n), ensuring efficient operations. AVL树的最大高度为O(log n)，保证了操作的高效性。

平衡因子/左重/右重 Balance Factor/Left-Heavy/Right-Heavy

- **AVL树的插入和删除**: 在AVL树中插入或删除元素的过程与普通二叉搜索树相同。 The process for inserting or deleting an element in an AVL tree is the same as in a regular binary search tree.

- **平衡因子**: 节点的平衡因子是其右子树高度减去左子树高度。 The balance factor of a node is the height of its right subtree minus the height of its left subtree.
- **左重/右重**: 当节点的平衡因子为-1时，节点称为左重；平衡因子为+1时，称为右重。 A node is said to be left-heavy if its balance factor is -1 and right-heavy if its balance factor is +1.

平衡树 Balancing Trees

- **重新平衡**: 如果插入或删除操作后节点不平衡（即平衡因子不是-1, 0, 或1），需要进行旋转以重新平衡。 If a node is not balanced (i.e., its balance factor is not -1, 0, or 1) after an insertion or deletion operation, you need to rebalance it.
- **四种旋转方式**: There are four possible rotations:
  - **LL旋转**（左重左重旋转） LL rotation (left-heavy left-heavy rotation)
  - **RR旋转**（右重右重旋转） RR rotation (right-heavy right-heavy rotation)
  - **LR旋转**（左重右重旋转） LR rotation (left-heavy right-heavy rotation)
  - **RL旋转**（右重左重旋转） RL rotation (right-heavy left-heavy rotation)

LL 失衡和 LL 旋转 LL Imbalance and LL Rotation

- **情况**: 当节点A的平衡因子为-2（左重），且其左子节点B的平衡因子为-1（左重）或0时，发生左重左重失衡。 An LL imbalance occurs at a node A if A has a balance factor of -2 (left-heavy) and its left child B has a balance factor of -1 (left-heavy) or 0.
- **LL旋转**: 单右旋转。 LL Rotation: a single right rotation.

```
/** Balance LL */
private void balanceLL(TreeNode<E> A, TreeNode<E> parentOfA) {
    TreeNode<E> B = A.left; // A is left-heavy and B is left-heavy
    if (A == root) {
        root = B;
    } else {
        if (parentOfA.left == A) {
            parentOfA.left = B;
        } else {
            parentOfA.right = B;
        }
    }
    A.left = B.right; // Make T2 the left subtree of A
    B.right = A; // Make A the right child of B
    updateHeight((AVLTreeNode<E>) A);
    updateHeight((AVLTreeNode<E>) B);
}
```

RR 失衡和 RR 旋转 RR Imbalance and RR Rotation

- **情况**: 当节点A的平衡因子为+2（右重），且其右子节点B的平衡因子为+1（右重）或0时，发生右重右重失衡。 An RR imbalance occurs at a node A if A has a balance factor of +2 (right-heavy) and its right child B has a balance factor of +1 (right-heavy) or 0.
- **RR旋转**: 单左旋转。 RR Rotation: a single left rotation.

```
/** Balance RR */
private void balanceRR(TreeNode<E> A, TreeNode<E> parentOfA) {
    TreeNode<E> B = A.right; // A is right-heavy and B is right-heavy
    if (A == root) {
        root = B;
    } else {
        if (parentOfA.left == A) {
            parentOfA.left = B;
        } else {
```

```
            parentOfA.right = B;
        }
    }
    A.right = B.left; // Make T2 the right subtree of A
    B.left = A;
    updateHeight((AVLTreeNode<E>) A);
    updateHeight((AVLTreeNode<E>) B);
}
```

LR 失衡和 LR 旋转 LR Imbalance and LR Rotation

- **情况**: 当节点A的平衡因子为-2（左重），且其左子节点B的平衡因子为+1（右重）时，发生左重右重失衡。 An LR imbalance occurs at a node A if A has a balance factor of -2 (left-heavy) and a left child B has a balance factor of +1 (right-heavy).
- **LR旋转**: 先对B进行一次左旋，再对A进行一次右旋。 This imbalance can be fixed by performing a double rotation: first a single left rotation at B and then a single right rotation at A.

```
/** Balance LR */
private void balanceLR(TreeNode<E> A, TreeNode<E> parentOfA) {
    TreeNode<E> B = A.left; // A is left-heavy
    TreeNode<E> C = B.right; // B is right-heavy
    if (A == root) {
        root = C;
    } else {
        if (parentOfA.left == A) {
            parentOfA.left = C;
        } else {
            parentOfA.right = C;
        }
    }
    A.left = C.right; // Make T3 the left subtree of A
    B.right = C.left; // Make T2 the right subtree of B
    C.left = B;
    C.right = A;
    // Adjust heights
    updateHeight((AVLTreeNode<E>) A);
    updateHeight((AVLTreeNode<E>) B);
    updateHeight((AVLTreeNode<E>) C);
}
```

RL 失衡和 RL 旋转 RL Imbalance and RL Rotation

- **情况**: 当节点A的平衡因子为+2（右重），且其右子节点B的平衡因子为-1（左重）时，发生右重左重失衡。 An RL imbalance occurs at a node A if A has a balance factor of +2 (right-heavy) and a right child B has a balance factor of -1 (left-heavy).
- **RL旋转**: 先对B进行一次右旋，再对A进行一次左旋。 This imbalance can be fixed by performing a double rotation: first a single right rotation at B and then a single left rotation at A.

```
/** Balance RL */
private void balanceRL(TreeNode<E> A, TreeNode<E> parentOfA) {
    TreeNode<E> B = A.right; // A is right-heavy
    TreeNode<E> C = B.left; // B is left-heavy
    if (A == root) {
        root = C;
    } else {
        if (parentOfA.left == A) {
            parentOfA.left = C;
        } else {
```

```
        parentOfA.right = C;
    }
}
A.right = C.left; // Make T2 the right subtree of A
B.left = C.right; // Make T3
```

为什么使用哈希？

1. 高效的数据管理：

- **集合 (Set)：** 哈希确保集合中的元素唯一且无序。这种结构使得检查元素是否存在、插入和删除操作都非常快速。
- **映射 (Map)：** 哈希使得可以快速查找与键关联的值。映射存储键值对，允许高效地检索给定键对应的值。

2. 性能优势：

- **O(1) 时间复杂度：** 哈希在查找、插入和删除操作上具有常数时间复杂度，远优于平衡搜索树的 O(log n) 复杂度。

哈希如何实现？

1. 数组的概念：

- 数组允许通过索引直接访问元素。这一概念在哈希表中得到利用，键被映射到索引，以快速访问或更新值。

2. 哈希表和哈希函数：

- **哈希表：** 存储值的数组。
- **哈希函数：** 将键映射到哈希表索引的函数。这涉及两个步骤：
    1. 将键转换为哈希码（整数）。
    2. 将哈希码压缩到哈希表的索引范围内。

哈希函数和哈希码

1. 生成哈希码：

- **转换为整数：** 哈希函数将搜索键转换为整数哈希码。
- **压缩：** 将哈希码缩小到哈希表索引范围（0到N-1）。

2. 示例和最佳实践：

- **Java 的 `hashCode` 方法：** 使用对象的内存地址转换为哈希码。
- **重写 `hashCode` ：** 确保两个相等的对象返回相同的哈希码，以保持一致性。
- **原始类型：** 使用 `Float.floatToIntBits` 、 `Double.doubleToLongBits` 和对 `long` 类型的位操作生成哈希码。
- **字符串：** 根据字符值和位置生成累积哈希码，确保哈希生成全面。

处理冲突

1. 定义和策略：

- **冲突：** 发生在两个不同的键映射到相同的索引时。
- **策略：**
    - **开放寻址法：** 使用线性探测、二次探测和双重散列等技术寻找下一个可用位置。
    - **分离链接法：** 将所有具有相同索引的条目存储在一个列表（或桶）中。

2. 开放寻址技术：

- **线性探测：** 顺序检查下一个槽，直到找到可用的槽。
- **二次探测：** 根据二次公式检查槽位，以减少聚集。
- **双重散列：** 使用二次哈希函数确定步长，进一步减少聚集。

负载因子和再哈希

1. 负载因子 (λ)：

- 元素数量 (n) 与哈希表大小 (N) 的比率。它表示哈希表的填充程度。

2. 再哈希：

- 当负载因子超过阈值（通常为0.75）时，哈希表会重新调整大小，并将所有现有条目重新哈希到新的更大的表中。这保持了效率，但是一项耗费资源的操作。

处理冲突

1. 定义和策略：

- **冲突：** 发生在两个不同的键映射到相同的索引时。这会导致无法直接存储新元素，必须采用处理冲突的策略。
- **策略：**
  - **开放寻址法：** 在哈希表内寻找另一个可用位置存储冲突的元素。主要有以下三种方法：
    - **线性探测：** 顺序检查下一个槽，直到找到可用的槽。
    - **二次探测：** 根据二次公式检查槽位，以减少聚集。
    - **双重散列：** 使用二次哈希函数确定步长，进一步减少聚集。
  - **分离链接法：** 将所有具有相同索引的条目存储在一个列表（或桶）中，通常采用链表或平衡树的方式。

2. 开放寻址技术：

- **线性探测：**
  - 当冲突发生时，从冲突位置开始，逐个检查下一个位置，直到找到一个空闲槽。例如，如果冲突发生在 `hashTable[key % N]`，则检查 `hashTable[(key + 1) % N]`，依次类推，直到找到空位置。
  - 删除元素时，需将该位置标记为“已删除”，而不是直接清空，以防止断开线性探测链。
- **二次探测：**
  - 使用二次公式 `(key + j^2) % N` 进行位置探测。例如，从 `key % N` 开始，依次检查 `(key + 1^2) % N`，`(key + 2^2) % N`，以此类推。
  - 这种方法可以减少线性探测中相邻元素聚集的情况，但仍可能产生次级聚集。
- **双重散列：**
  - 使用两个不同的哈希函数 `h1(key)` 和 `h2(key)`。初始位置由 `h1(key)` 确定，探测步长由 `h2(key)` 确定。例如，从 `h1(key)` 开始，依次检查 `(h1(key) + j * h2(key)) % N`。
  - 这种方法有效地减少了聚集问题，因为探测步长由键的第二个哈希值决定，增加了位置分散性。

负载因子和再哈希

1. 负载因子 (λ)：

- 负载因子是元素数量 (n) 与哈希表大小 (N) 的比率。公式为 `λ = n / N`。
- 负载因子表示哈希表的填充程度。当负载因子增加时，冲突的概率也增加。

2. 再哈希：

- 当负载因子超过预定阈值（例如 0.75）时，哈希表需要进行再哈希操作：
  - **增大哈希表：** 通常是将哈希表的大小增加到当前的两倍或更大。
  - **重新分配元素：** 将所有现有的元素重新计算哈希值并放入新的更大的哈希表中。
  - **调整哈希函数：** 由于哈希表的大小已更改，需要修改哈希函数以适应新的大小。

再哈希策略：

- **保持性能：** 虽然再哈希是一个耗时的操作，但通过增大哈希表，减少了冲突的发生，保证了插入、删除和查找操作的高效性。
- **成本管理：** 选择合适的阈值和扩展策略，尽量减少再哈希的频率。例如，Java 的 `HashMap` 使用 0.75 作为阈值，并在负载因子超过此阈值时将哈希表大小加倍。

详细示例：

线性探测示例：

- 假设哈希表大小为 11，初始哈希函数为 `h(key) = key % 11`。
- 插入键 `12`：`h(12) = 1`，如果位置 1 已被占用，则检查位置 `2 (1+1)`，再检查 `3 (1+2)`，以此类推。
- 插入键 `23`：`h(23) = 1`，同样从位置 1 开始探测，依次检查位置 2、3，直到找到空位置。

二次探测示例：

- 同样哈希表大小为 11，初始哈希函数为 `h(key) = key % 11`。
- 插入键 `12`：`h(12) = 1`，如果位置 1 已被占用，则检查位置 `2 (1+1^2)`，再检查位置 `5 (1+2^2)`，以此类推。
- 插入键 `23`：`h(23) = 1`，从位置 1 开始探测，依次检查位置 2、5，直到找到空位置。

双重散列示例：

- 哈希表大小为 11，初始哈希函数为 `h1(key) = key % 11` 和 `h2(key) = 7 - (key % 7)`。
- 插入键 `12`：`h1(12) = 1`，`h2(12) = 2`，从位置 1 开始探测，依次检查位置 `3 (1+1*2)`、`5 (1+2*2)`，以此类推。
- 插入键 `23`：`h1(23) = 1`，`h2(23) = 5`，从位置 1 开始探测，依次检查位置 `6 (1+1*5)`、`0 (1+2*5)`，直到找到空位置。

通过以上策略和方法，可以有效处理哈希冲突，保证哈希表的高效运作。

处理冲突的策略比较 (Handling Collisions in Hashing)

处理哈希冲突的主要策略有两种：**开放寻址法**和**分离链接法**。它们在解决冲突、内存使用、性能等方面各有优缺点。下面是这两种策略的详细区别：

The two main strategies for handling hash collisions are **Open Addressing** and **Separate Chaining**. They differ in terms of collision resolution, memory usage, performance, and other aspects. Here are the detailed differences between these two strategies:

1. 开放寻址法 (Open Addressing)

基本概念 (Basic Concept):

- 当发生冲突时，在哈希表内寻找另一个可用位置存储冲突的元素，而不是直接存储在冲突位置。
- When a collision occurs, another available position within the hash table is searched to store the conflicting element, rather than storing it at the collision position.

方法 (Methods):

- **线性探测 (Linear Probing):** 顺序检查下一个槽，直到找到可用的槽。
  - Sequentially check the next slot until an available slot is found.
- **二次探测 (Quadratic Probing):** 根据二次公式检查槽位，以减少聚集。
  - Check slots based on a quadratic formula to reduce clustering.
- **双重散列 (Double Hashing):** 使用二次哈希函数确定步长，进一步减少聚集。
  - Use a secondary hash function to determine the step size, further reducing clustering.

优点 (Advantages):

- **内存紧凑 (Memory Efficient):** 所有元素存储在哈希表内部，无需额外的存储空间。
  - All elements are stored within the hash table, requiring no extra storage space.
- **简洁 (Simple):** 实现相对简单，不需要额外的数据结构（如链表或树）。
  - Relatively simple implementation without the need for additional data structures like linked lists or trees.

缺点 (Disadvantages):

- **聚集问题 (Clustering):** 线性探测会产生主要聚集，二次探测会产生次要聚集，双重散列可以缓解但不能完全避免。
  - Linear probing causes primary clustering, quadratic probing causes secondary clustering, and double hashing mitigates but does not entirely avoid clustering.
- **删除复杂 (Complex Deletion):** 删除元素需要标记“已删除”而不是直接清空，以维护探测链的完整性。
  - Deletion requires marking as "deleted" rather than clearing to maintain the probe chain's integrity.
- **装载因子限制 (Load Factor Limit):** 装载因子较高时，冲突增加，性能下降。通常装载因子需保持在0.7以下。
  - High load factors increase collisions and degrade performance, typically requiring a load factor below 0.7.

性能影响 (Performance Impact):

- 插入、删除和查找的最坏情况下时间复杂度为O(n)，平均情况下接近O(1)。

    - Worst-case time complexity for insertion, deletion, and search is O(n), with average cases close to O(1).
- 冲突处理复杂度随装载因子增加而增加。
    - Collision handling complexity increases with the load factor.

2. 分离链接法 (Separate Chaining)

基本概念 (Basic Concept):

- 将所有具有相同哈希索引的条目存储在一个列表（桶）中，每个桶可以是链表、平衡树或其他结构。
- Store all entries with the same hash index in a list (or bucket), where each bucket can be a linked list, balanced tree, or other structure.

方法 (Methods):

- **链表 (Linked List):** 最常用的形式，每个哈希表索引处存储一个链表，链表中的每个节点包含一个键值对。
    - The most common form, where each hash table index stores a linked list, and each node in the list contains a key-value pair.
- **平衡树 (Balanced Tree):** 在冲突频繁的情况下，可以将链表替换为平衡树，以优化查找性能。
    - Replace the linked list with a balanced tree in cases of frequent collisions to optimize lookup performance.

优点 (Advantages):

- **冲突分离 (Independent Collisions):** 不同哈希索引的元素冲突彼此独立，冲突处理简单直接。
    - Collisions are independent for different hash indices, making collision handling simple and direct.
- **装载因子弹性 (Flexible Load Factor):** 装载因子可以超过1，因为每个桶可以存储多个元素。
    - The load factor can exceed 1, as each bucket can store multiple elements.

缺点 (Disadvantages):

- **额外内存 (Extra Memory):** 需要额外的内存来存储链表节点或树节点。
    - Requires extra memory to store linked list or tree nodes.
- **复杂性 (Complexity):** 链表和树的管理增加了代码复杂性。
    - Managing linked lists and trees increases code complexity.

性能影响 (Performance Impact):

- 插入、删除和查找的最坏情况下时间复杂度为O(n)，但通常情况下，随着装载因子增加，性能依然可以保持较好。
    - Worst-case time complexity for insertion, deletion, and search is O(n), but typically, performance remains good as the load factor increases.
- 平均情况下，时间复杂度接近O(1)，尤其在装载因子合理时。
    - Average case time complexity is close to O(1), especially when the load factor is reasonable.

总结对比 (Summary Comparison)

特性 (Feature)	开放寻址法 (Open Addressing)	分离链接法 (Separate Chaining)
内存使用 (Memory Usage)	紧凑，无需额外空间	需要额外空间存储链表或树节点
实现复杂性 (Implementation Complexity)	简单，不需要额外的数据结构	复杂，需要管理链表或树
冲突处理 (Collision Handling)	探测（线性、二次或双重散列）	链表或树结构
聚集问题 (Clustering Issue)	存在主要和次要聚集	无聚集问题
删除操作 (Deletion Operation)	复杂，需要标记“已删除”状态	简单，直接删除链表或树节点
装载因子 (Load Factor)	需保持在0.7以下，装载因子高时性能下降	可以超过1，装载因子对性能影响较小
性能 (Performance)	平均情况下接近O(1)，最坏情况下O(n)，受装载因子影响较大	平均情况下接近O(1)，最坏情况下O(n)，受装载因子影响较小
再哈希需求 (Rehashing Needs)	装载因子高时需频繁再哈希	再哈希需求较低

通过以上对比，可以看出开放寻址法适用于对内存使用要求较高且装载因子低的场景，而分离链接法则更适用于可能出现高装载因子的情况，尤其是在需要处理大量冲突的场合。选择哪种策略主要取决于具体应用场景和性能需求。

From the comparison above, it can be seen that open addressing is suitable for scenarios requiring high memory efficiency and low load factors, whereas separate chaining is more suitable for situations with potentially high load factors, especially when handling a large number of collisions. The choice of strategy depends mainly on the specific application scenario and performance requirements.

bucket三种状态

在开放寻址法中，每个槽（bucket）有三种状态的原因是为了有效地管理插入、搜索和删除操作。这三种状态分别是：占用（occupied）、标记（marked，也称为“删除标记”）和空闲（empty）。下面详细解释每种状态及其重要性：

三种状态的定义和重要性

1. **占用状态 (Occupied)**
   - **定义：** 该槽已经存储了一个有效的键值对。
   - **重要性：** 在搜索时，如果找到一个占用的槽且键匹配，则搜索成功。在插入时，碰到占用的槽需要继续探测下一个槽。
2. **标记状态 (Marked)**
   - **定义：** 该槽曾经存储过一个键值对，但已被删除。
   - **重要性：** 标记状态对于搜索和插入操作都非常重要。
     - **搜索：** 搜索操作时，碰到标记状态的槽不能立即停止搜索，因为目标键可能在后续的槽中。如果直接跳过标记状态的槽，可能会漏掉实际存在的键。
     - **插入：** 插入操作时，碰到标记状态的槽可以将新的键值对插入其中，重新利用这个槽。
   - **删除：** 通过标记而不是直接清空槽，可以确保搜索路径完整，不会导致搜索提前终止。
3. **空闲状态 (Empty)**
   - **定义：** 该槽从未存储过任何键值对。
   - **重要性：** 在搜索时，如果碰到空闲状态的槽，表示搜索结束且键不存在。在插入时，碰到空闲状态的槽即可直接插入新键值对。

示例代码

以下是一个使用三种状态的开放寻址法示例代码：

```
import java.util.Arrays;

class OpenAddressingHashTable {
    private static final int EMPTY = -1;
    private static final int MARKED = -2;
    private int[] hashTable;
    private int size;

    public OpenAddressingHashTable(int size) {
        this.size = size;
        this.hashTable = new int[size];
        Arrays.fill(hashTable, EMPTY); // Initialize all slots to empty
    }

    private int hashFunction(int key) {
        return key % size;
    }

    public void insert(int key) {
        int hashValue = hashFunction(key);
        while (hashTable[hashValue] != EMPTY && hashTable[hashValue] != MARKED) {
```

```
            hashValue = (hashValue + 1) % size;
        }
        hashTable[hashValue] = key;
    }

    public boolean search(int key) {
        int hashValue = hashFunction(key);
        while (hashTable[hashValue] != EMPTY) {
            if (hashTable[hashValue] == key) {
                return true;
            }
            hashValue = (hashValue + 1) % size;
        }
        return false;
    }

    public void delete(int key) {
        int hashValue = hashFunction(key);
        while (hashTable[hashValue] != EMPTY) {
            if (hashTable[hashValue] == key) {
                hashTable[hashValue] = MARKED; // Mark as deleted
                return;
            }
            hashValue = (hashValue + 1) % size;
        }
    }
}
```

例子分析 (Example Analysis)

1. **插入 (Insert)**
   - 将键 `5` 插入哈希表时，计算得到的索引是 `5 % size`。
   - 如果槽 `5` 是空闲的，则直接插入。如果槽 `5` 已被占用，则检查下一个槽，直到找到一个空闲或标记的槽。
2. **搜索 (Search)**
   - 搜索键 `5` 时，从索引 `5 % size` 开始。
   - 遇到标记槽或占用槽时，继续搜索；遇到空闲槽时停止搜索（键不存在）。
3. **删除 (Delete)**
   - 删除键 `5` 时，从索引 `5 % size` 开始，找到匹配的键后，将槽状态标记为删除。

结论 (Conclusion)

开放寻址法中的三种状态（占用、标记、空闲）确保了插入、搜索和删除操作的正确性和效率。标记状态特别重要，因为它允许哈希表重新使用已删除的槽，并维护正确的搜索路径，从而提高哈希表的整体性能和存储利用率。

The three states (occupied, marked, empty) in open addressing ensure correctness and efficiency in insert, search, and delete operations. The marked state is particularly important as it allows the hash table to reuse deleted slots and maintain correct search paths, thereby enhancing the overall performance and storage utilization of the hash table.