

```

public static int[] revert(int[] input) {
    if (input == null) {
        return null;
    }
    int length = input.length;
    int[] output = new int[length];
    for (int index = 0; index < length; index++) {
        output[length - 1 - index] = input[index];
    }
    return output;
}

```

```

public class BankAccount {

    private final long number;
    private long balance;

    public BankAccount(long number) {
        this.number = number;
    }

    public boolean deposit(long amount) {
        if (amount < 0) {
            return false;
        }
        balance += amount;
        return true;
    }

    public boolean withdraw(long amount) {
        if (amount < 0 || balance - amount < 0) {
            return false;
        }
        balance -= amount;
        return true;
    }

    public long getNumber() {
        return number;
    }

    public long getBalance() {
        return balance;
    }
}

```

```

public interface Shape extends GraphicItem {
    int getAreaColor();
    void draw(Visualization visualization);
}

```

```

public class Circle implements Shape {

    private int centerX;
    private int centerY;
    private int radius;
    private int color;

    public Circle(int centerX, int centerY, int radius, int color) {
        this.centerX = centerX;
        this.centerY = centerY;
        this.radius = radius;
        this.color = color;
    }

    public int getRadius() {
        return radius;
    }

    @Override
    public int getAreaColor() {
        return color;
    }

    @Override
    public int getX() {
        return centerX;
    }
}

```

```

private List<BankAccount> allAccounts() {
    var allAccounts = new ArrayList<BankAccount>();
    for (var customer : customers) {
        var customerAccounts = customer.getAccounts();
        for (BankAccount account : customerAccounts) {
            allAccounts.add(account);
        }
    }
    return allAccounts;
}

```

```

public class Folder extends Element {

    private final List<Element> entries = new ArrayList<>();

    public Folder(String name) {
        super(name);
    }

    public void addEntry(Element element) {
        entries.add(element);
    }

    @Override
    public long getSize() {
        long totalSize = 0L;
        for (Element entry : entries) {
            totalSize += entry.getSize();
        }
        return totalSize;
    }

    //task2
    public void accept(FileSystemVisitor visitor) {
        visitor.visit(this);
        for (Element entry : entries) {
            entry.accept(visitor);
        }
        visitor.leave(this);
    }
}

```

```

//a) Abhängigkeit Bank zu BankCustomer
private ArrayList<BankCustomer> customers = new ArrayList<>();

```

```

public class Stack<T> {

    private ArrayList<T> elements = new ArrayList<>();

    public void push(T element) {
        elements.add(element);
    }

    public T pop() {
        if (isEmpty()) {
            return null;
        }
        int lastIndex = elements.size() - 1;
        T element = elements.remove(lastIndex);
        return element;
    }

    public int size() {
        return elements.size();
    }

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

```

```

public class ExceptionTheory {
    /**
     * x == 0: "Finally", verlässt Methode mit Exception
     *
     * x == 1: "Second catch", "Finally", verlässt Methode normal (ohne Exception)
     *
     * x == 2: "First catch", "Finally", verlässt Methode normal
     *
     * x sonst: "Finally", verlässt Methode normal
     */
    void test1(int x) throws Exception {
        try {
            if (x == 0) {
                throw new Exception();
            } else if (x == 1) {
                throw new MyException();
            } else if (x == 2) {
                throw new MySubException();
            }
        } catch (MySubException e) {
            System.out.println("First catch");
        } catch (MyException e) {
            System.out.println("Second catch");
        } finally {
            System.out.println("Finally");
        }
    }
}

```

```

// a)
public static List<String> getDistinctThreeLetterFemaleFirstnames(List<Person> people) {
    return people.stream().filter(p -> p.getGender() == Gender.FEMALE).map(Person::getFirstName)
        .filter(n -> n.length() <= 3).distinct().collect(Collectors.toList());
}

// b)
public static double getAverageMaleAge(List<Person> people) {
    return people.stream().filter(p -> p.getGender() == Gender.MALE).mapToInt(Person::getAge).average()
        .getAsDouble();
}

// c)
public static int getMinAge(List<Person> people) {
    return people.stream().mapToInt(Person::getAge).min().getAsInt();
}

```

```

try (var reader = new BufferedReader(new InputStreamReader(in, Charset.forName("UTF-8")))) {

    String line;
    while ((line = reader.readLine()) != null) {
        var matcher = pattern.matcher(line);
        while (matcher.find()) {
            var word = matcher.group().toLowerCase();

            // If we see this word for the first time, add it with the count 1, otherwise we
            // just increment the count by one:
            if (!histogram.containsKey(word)) {
                histogram.put(word, 1L);
            } else {
                histogram.put(word, histogram.get(word) + 1);
            }

            // Das if/else könnte auch durch folgende Methode ersetzt werden:
            // histogram.merge(word, 1, (v1, v2) -> v1 + v2);
        }
    }
}

```

```

public static void sort(List<Person> people) throws IOException {
    people.sort((p1, p2) -> Integer.compare(p1.getAge(), p2.getAge()));
    PeopleData.write("sorted2AgeAsc.csv", people);
    people.sort((p1, p2) -> Integer.compare(p2.getAge(), p1.getAge()));
    PeopleData.write("sorted2AgeDesc.csv", people);
    people.sort((p1, p2) -> p1.getLastName().compareTo(p2.getLastName()));
    PeopleData.write("sorted2LastNames.csv", people);
    people.sort((p1, p2) -> {
        String name1 = p1.getFirstName() + p1.getLastName();
        String name2 = p2.getFirstName() + p2.getLastName();
        return Integer.compare(name2.length(), name1.length());
    });
}

```

```

public class TemperatureConverter extends JFrame {

    private static final long serialVersionUID = 1L;
    private JPanel contentPane;

    public TemperatureConverter() {
        setTitle("Temperature Converter");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        contentPane = new JPanel();
        setContentPane(contentPane);
        contentPane.setLayout(new GridLayout(3, 3));

        JLabel promptLabel = new JLabel("Degrees");

        JTextPane inputText = new JTextPane();
        contentPane.add(inputText);

        ButtonGroup buttonGroup = new ButtonGroup();

        JRadioButton celsiusRadio = new JRadioButton("Celcius");
        buttonGroup.add(celsiusRadio);

        JRadioButton fahrenheitRadio = new JRadioButton("Fahrenheit");
        buttonGroup.add(fahrenheitRadio);

        JTextPane outputText = new JTextPane();
        outputText.setEditable(false);

        JButton convertButton = new JButton("Convert");
        convertButton.addActionListener(actionEvent -> {
            try {
                double temp = Double.parseDouble(inputText.getText());
                if (celsiusRadio.isSelected()) {
                    temp = temp * 1.8 + 32;
                } else if (fahrenheitRadio.isSelected()) {
                    temp = (temp - 32) * (5.0 / 9.0);
                }
                outputText.setText(String.format("%.2f", temp));
            } catch (NumberFormatException e) {
                outputText.setText("Please enter a valid temperature.");
            }
        });

        contentPane.add(promptLabel);
        contentPane.add(inputText);

        contentPane.add(celsiusRadio);
        contentPane.add(fahrenheitRadio);

        contentPane.add(convertButton);
        contentPane.add(outputText);

        pack();
        setVisible(true);
    }
}

```

Map

Abbildung Schlüssel → Werte

```

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

Student st1 = new Student("Andrea", "Meier");
Student st1 = new Student("Bertha", "Mueller");
Student st1 = new Student("Clara", "Schneider");

map.put(20000, st1);    // Bei gleichem Schluessel
map.put(70000, st2);    // wird der Wert ueberschrieben
map.put(13000, st3);    // -> nur ein Key pro Map

Student st = map.get(12345);    // Finden nach Schluessel (sehr effizient)

for(Student s : map.values()){ // .values() gibt Collection aller Werte
    System.out.println(s);
}

```

Lambdas

Ein Lambda ist eine Referenz auf eine **anonyme** Methode. Der Syntax sieht wie folgt aus:

```
(Parameterliste) -> { Body }
```

Beispiel:

```
(p1, p2) -> { return Integer.compare(p1.getAge(), p2.getAge()); }
```

Dabei sind die geschweiften Klammern im Body optional, werden nur bei einem syntaktischer Ausdruck (Expression) gebraucht.

```
(p1, p2) -> Integer.compare(p1.getAge(), p2.getAge())
```

Bei nur einem Übergabeparameter sind sogar die runden Klammern überflüssig.

```
p -> p.getAge() >= 18
```

letzterer Ausdruck ist Analog zu **this::isAdult** mit:

```
boolean isAdult(Person p){  
    return p.getAge() >= 18;  
}
```

Lambdas sind auch ohne Parameterliste möglich, was dann wie folgt aussieht:

```
() -> System.out.println("Do nothing!");
```

Das benützen eines Lambdas und das Functional Interface sieht dann wie folgt aus:

```
public void testAll(Predicate<T> criterion) {  
    List<Integer> integerList =  
        Arrays.asList(10, 20, 18, 21);  
    for(int elem: integerList){  
        if(criterion.test(elem)){  
            System.out.println(elem + '\n');  
        }  
    }  
}
```

```
@FunctionalInterface  
public interface Predicate<T> {  
    boolean test(T element);  
}
```

@FunctionalInterface ist nicht nötig, aber kann zur Information hingeschrieben werden. Für den Compiler ändert sicher aber nichts.

Beginn

Begonnen wird immer mit dem Aufruf von `stream()` bei Collection. Beispiel:

```
people.stream()
```

Method Chaining / fluent Interface

```
people  
    .stream()  
    .filter(p -> p.getAge() >= 18)  
    .map(p -> p.getLastName())  
    .sorted()  
    .forEach(System.out::println);
```

Ist äquivalent zu ↓

```
Stream<Person> stream1 = people.stream();  
Stream<Person> stream2 = stream1.filter(p -> p.getAge() >= 18);  
Stream<Person> stream3 = stream2.map(p -> p.getLastName());  
Stream<Person> stream4 = stream3.sorted();  
stream4.forEach(s -> System.out.println(s));
```



Zwischenoperationen

```
filter(Predicate)    // Rauspicken gemäss Predicate-Funktionsobjekt/Lambda  
map(Function)        // Projizieren gemäss Funktionsobjekt/Lambda  
mapToInt(Function)   // Projizieren auf int, long double (primitiver Datentyp)  
sorted()             // sortieren, mit oder ohne Comparator  
distinct()           // keine gleichen Elemente gemässss Equals()  
limit(long n)        // erste n Elemente liefern, danach ignorieren  
skip(long n)         // erste n Elemente ignorieren, danach weiterliefern
```

Terminaloperationen

```
forEach(Consumer)    // Pro Element Operation anwenden, meist mit Seiteneffekt  
count()              // Anzahl Element  
min(), max()         // Mit Comparator Argument  
average(), sum()     // Nur bei Int/Long/Double Streams  
findAny(), findFirst() // Gibt irgendein bzw. erstes Element zurück
```

