1. Да се напише метод кој враќа точно ако `String str1` е префикс на `String str2`. Не е дозволено користење на готови методи за пребарување, единствено дозволено е користење на `charAt`.

```
public static boolean isPrefix(String str1, String str2) {
    if(str1.length() > str2.length()) {
        return false;
    }
    for(int i = 0; i < str1.length(); i++) {
        if(str1.charAt(i) != str2.charAt(i)) {
            return false;
        }
    }
    return true;
}
```

2. Да се имплементираат следните методи кои примаат дво-димензионални низи од double и враќаат како резултат сума и просек.

```
public static double sum(double[][] a)
public static double average(double[][] a)
```

```
public static double sum(double[][] a) {
    double s = 0;
    for(int i = 0; i < a.length; i++) {
        for(int j = 0; j < a[i].length; j++) {
            s += a[i][j];
        }
    }
    return s;
}

public static double avarage(double[][] a) {
    double s = 0;
    for(int i = 0; i < a.length; i++) {
        for(int j = 0; j < a[i].length; j++) {
            s += a[i][j];
        }
    }
    return s / (a.length * a[0].length);
}
```

3. Катанец со комбинации (Combination Lock) ги има следните својства:

   - комбинацијата (секвенца од 3 цифри) е скриена
   - катанецот може да се отвори само ако се внесе точната комбинација
   - комбинацијата може да се промени само ако се знае претходната комбинација.

   Да се дизајнира класа со јавни методи `open` и `changeCombo` и приватни податоци кои ја чуваат комбинацијата. Комбинацијата се поставува преку конструкторот.

```
package edu.finki.np.av2;

public class CombinationLock {
```

```
    private int combination;
    private boolean isOpen;

    public CombinationLock(int combination) {
        this.combination = combination;
        this.isOpen = false;
    }

    public boolean open(int combination) {
        this.isOpen = (this.combination == combination);
        return this.isOpen;
    }

    public boolean changeCombo(int combination, int
        newCombination) {
        boolean isCorrect = (this.combination == combination);
        if (isCorrect) {
            this.combination = newCombination;
        }
        return isCorrect;
    }

}
```

4. Имплементирајте едноставна класа за датум (Date). Вашата класа треба да може:

   - да ги репрезентира сите датуми од 1 Јануари 1800 до 31 Декември 2500
   - одзема два датуми
   - зголеми датум за одреден ден денови
   - да споредува два датуми со помош на `equals` и `compareTo`.

   Датумот внатрешно ќе се репрезентира како број на денови од почетното време, кое во овој случај е почетокот на 1800. Со ова сите методи освен методот `toString` се поедноставуваат. Да се запази правилото за престапни години (престапна година е секоја година која е делива со 4 и не е делива со 100 освен ако е делива со 400). Конструкторот треба да ја провери валидноста на датумот, а исто така и методот `toString`. Датумот може да стане неточен ако зголемувањето или намалувањето придонсе да излезе надвор од опсегот.

   Потешкиот дел од задачата е конверзијата од интерната репрезентација во надворешна репрезентација на датум. Еден предлог алгоритам е следниот. Се иницијализираат две низи во статички членови. Првата низа е денови до први во месецот (daysTillFirstOfMonth) во не престапна година. Оваа низа содржи 0, 31, 59, 90, итн. Во втората низа, денови од почетокот на првата година (daysTillJan1). Така оваа низа ќе содржи 0, 365, 730, 1095, 1460, 1826, итн. Со помош на овие низи ќе ја правиме конверзијата на различни репрезентации на датум.

```
package edu.finki.np.av2;

public class Date {
    private static final int FIRST_YEAR = 1800;
    private static final int LAST_YEAR = 2500;

    private static final int[] daysOfMonth = {
```

```
        31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
    };
    private static final int[] daysTillFirstOfMonth;
    private static final int[] daysTillJan1;
    static {
        daysTillFirstOfMonth = new int[12];
        for(int i = 1; i < 12; i++) {
            daysTillFirstOfMonth[i] += daysTillFirstOfMonth[i -
                1] + daysOfMonth[i - 1];
        }
        int totalYears = LAST_YEAR - FIRST_YEAR;
        daysTillJan1 = new int[totalYears + 1];
        int currentYear = FIRST_YEAR;
        for(int i = 1; i <= totalYears; i++) {
            if(isLeapYear(currentYear)) {
                daysTillJan1[i] = daysTillJan1[i - 1] + 366;
            } else {
                daysTillJan1[i] = daysTillJan1[i - 1] + 365;
            }
            currentYear++;
        }
    }

    private static boolean isLeapYear(int year) {
        return (year % 400 == 0 || (year % 4 == 0 && year % 100
            != 0));
    }

    private int days;

    public Date(int days) {
        this.days = days;
    }

    public Date(int date, int month, int year) {
        days = 0;
        if(year < FIRST_YEAR || year > LAST_YEAR) {
            throw new RuntimeException();
        }
        days += daysTillJan1[year - FIRST_YEAR];
        days += daysTillFirstOfMonth[month - 1];
        if(month > 2 && isLeapYear(year)) {
            days++;
        }
        days += date;
    }

    public int substract(Date date) {
        return this.days - date.days;
    }

    public Date increment(int days) {
        return new Date(this.days + days);
    }

    @Override
    public boolean equals(Object arg0) {
```

```
        Date date = (Date)arg0;
        return this.days == date.days;
    }

    public int compareTo(Date date) {
        return this.days - date.days;
    }

    @Override
    public String toString() {
        int d = days;
        int i;
        for(i = 0; i < daysTillJan1.length; i++) {
            if(daysTillJan1[i] >= days) {
                break;
            }
        }
        d -= daysTillJan1[i - 1];
        int year = FIRST_YEAR + i - 1;
        for(i = 0; i < daysTillFirstOfMonth.length; i++) {
            if(daysTillFirstOfMonth[i] >= d) {
                break;
            }
        }
        int month = i;
        d -= daysTillFirstOfMonth[i - 1];
        int date = d;
        if(isLeapYear(year)) {
            date--;
        }
        return String.format("%02d.%02d.%4d", date, month, year)
            ;
    }

    public static void main(String[] args) {
        Date sample = new Date(1, 10, 2012);
        System.out.println("Date: " + sample);
        System.out.println(sample.substract(new Date(1, 1, 2000)
            ));
        System.out.println(sample);
        sample = new Date(1, 1, 1800);
        System.out.println(sample);
        sample = new Date(31, 12, 2500);
        System.out.println(sample);
        sample = new Date(31, 12, 2300);
        System.out.println(sample);
        sample = sample.increment(100);
        System.out.println(sample);
    }

}
```

5. `PlayingCard` (карта) е класа со која се репрезентира карта во игри со карти како покер и блек џек и во не се чува информација за бојата (херц, каро, пик, треф) и рангот (вредност од 2 до 10 или џандар, кралица, поп или ас). `Deck` (шпил од карти) е класа која репрезентира комплет од 52 карти. Додека `MultipleDeck`

(повеќе шпилови) е класа која репрезентира повеќе шпилови со карти (точниот број се задава во конструкторот). Да се имплментираат 3 класи `PlayingCard`, `Deck`, и `MultipleDeck`, со стандардна функционалност за карта, за шпил и повеќе шпилови да се имплментираат методи за мешање `shuffle`, делење на карта `dealCard` и проверка дали има останато карти.

```
package edu.finki.np.av2;


public class PlayingCard {
    public static final int MAX_RANK = 13;
    public static final int MIN_RANK = 1;
    public enum SUIT {
        HEARTS,
        DIAMONDS,
        CLUBS,
        SPADES
    }

    private SUIT suit;
    private int rank;

    public PlayingCard(SUIT suit, int rank) {
        this.suit = suit;
        this.rank = rank;
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder(String.valueOf(rank
            ));
        if(this.suit == SUIT.HEARTS) {
            sb.append("H");
        } else if(this.suit == SUIT.DIAMONDS) {
            sb.append("D");
        } else if(this.suit == SUIT.CLUBS) {
            sb.append("C");
        } else if(this.suit == SUIT.SPADES) {
            sb.append("S");
        }
        return sb.toString();
    }
}
```

```
package edu.finki.np.av2;
import java.util.Random;

public class Deck {
    public static final int SIZE = 52;

    private PlayingCard[] cards;
    private boolean[] isUsed;
    private int totalUsed;

    public Deck() {
        cards = new PlayingCard[SIZE];
        int n = 0;
```

```
            for (PlayingCard.SUIT suit : PlayingCard.SUIT.values())
                {
                for (int rank = PlayingCard.MIN_RANK; rank <=
                    PlayingCard.MAX_RANK; rank++) {
                    cards[n * (PlayingCard.MAX_RANK - PlayingCard.
                        MIN_RANK + 1) + rank
                            - 1] = new PlayingCard(suit, rank);
                }
                n++;
            }
            isUsed = new boolean[SIZE];
            for(int i = 0; i < SIZE; i++) {
                isUsed[i] = false;
            }
            totalUsed = 0;
        }

        public PlayingCard deal() {
            if(totalUsed < SIZE) {
                Random random = new Random();
                int cardIndex =random.nextInt(SIZE);
                if(!isUsed[cardIndex]) {
                    isUsed[cardIndex] = true;
                    totalUsed++;
                    return cards[cardIndex];
                } else {
                    return deal();
                }
            }
            return null;
        }

        @Override
        public String toString() {
            StringBuilder sb = new StringBuilder();
            for(PlayingCard card : cards) {
                sb.append(card.toString());
                sb.append(" ");
            }
            return sb.toString();
        }

        public static void main(String[] args) {
            Deck deck = new Deck();
            System.out.println(deck);
            for(int i = 0; i < SIZE; i++) {
                System.out.println("DEAL: " + deck.deal());
            }
        }
}
```

```
package edu.finki.np.av2;

public class MultipleDeck {
    private Deck[] decks;

    public MultipleDeck(int size) {
```

```
            decks = new Deck[size];
            for(int i = 0; i < size; i++) {
                decks[i] = new Deck();
            }
        }
    }
```

6. Комплексен број се состои од реален дел и имагинарен дел. Да се импалементира класа `BigComplex`, во која реалниот и имагинарниот дел ќе се чуваат во објекти од класата `BigDecimals`.

```
package edu.finki.np.av2;
import java.math.BigDecimal;

public class BigComplex {
    private BigDecimal real;
    private BigDecimal imag;

    public BigComplex(BigDecimal real, BigDecimal imag) {
        this.real = real;
        this.imag = imag;
    }

    public BigComplex add(BigComplex complex) {
        BigDecimal real = this.real.add(complex.real);
        BigDecimal imag = this.real.add(complex.imag);
        return new BigComplex(real, imag);
    }

}
```

7. Дадени се следниве пет класи: `Bank`, `Account`, `NonInterestCheckingAccount`, `InterestCheckingAccount` и `PlatinumCheckingAccount`, како и интефејс наречен `InterestBearingAccount` кои се однесуваат на следниот начин:

   - Во `Bank` чува листа од сите видови сметки, вклучувајќи сметки за штедење и за трошење, некои од нив подложни на камата, а некои не. Во `Bank` постои метод `totalAssets` кој ја враќа сумата на состојбата на сите сметки. Исто така содржи метод `addInterest` кој го повикува методот `addInterest` на сите сметки кои се подложни на камата.
   - `Account` е апстрактна класа. Во секој сметка се чуваат името на сопственикот на сметката, бројот на сметката (секвенцијален број доделен автоматски), моменталната состојба. Во класата се имплементираат конструктор за иницијализација на податочните членови, методи за пристап до моменталната состојба, како и за додавање и одземање од моменталната состојба.
   - `InterestBearingAccount` интерфејсот декларрира единствен метод `addInterest` (без параметри и не враќа ништо - `void`) кој ја зголемува состојбата со соодветната камата за овој вид на сметка.
   - `InterestCheckingAccount` е сметка `Account` која е исто така `InterestBearingAccount`. Повикување `addInterest` ја зголемува состојбата за 3%.
   - `PlatinumCheckingAccount` е `InterestCheckingAccount`. Повикување `addInterest` ја зголемува состојбата двојно од каматата за `InterestCheckingAccount` (колку и да е таа).

- `NonInterestCheckingAccount` е сметка `Account` но не е `InterestBearingAccount`. Нема дополнителни функционалности надвор од основните од класата `Account`.

За оваа задача, треба да се направи следното. Потребно е да се имплментира функционалност дадена во претходниот текст:

а Пет од шест класи од споменатите формираат хиерархија. За овие класи да се нацрта оваа хиерархија.

б Да се имплементира `Account`.

в Да се имплементира `NonInterestCheckingAccount`.

г Да се напише `InterestBearingAccount` интерфејсот.

д Да се имплементира `Bank`.

е Да се имплементира `InterestCheckingAccount`.

ж Да се имплементира `PlatinumCheckingAccount`.

```java
package edu.finki.np.av2.bank;

public abstract class Account {
    private String holderName;
    private int number;
    private double currentAmount;

    public Account(String holderName, int number, double
        currentAmount) {
        this.holderName = holderName;
        this.number = number;
        this.currentAmount = currentAmount;
    }

    public double getCurrentAmount() {
        return currentAmount;
    }

    public void addAmount(double amount) {
        currentAmount += amount;
    }

    public void withdrawAmount(double amount) {
        currentAmount -= amount;
    }

}
```

Listing 1: Имплементација на класата `Account`

```java
package edu.finki.np.av2.bank;

public class NonInterestCheckingAccount extends Account {

    public NonInterestCheckingAccount(String holderName, int
        number,
             double currentAmount) {
        super(holderName, number, currentAmount);
    }
```

```
}
```

```
package edu.finki.np.av2.bank;

public interface InterestBearingAccount {
    public void addInterest();
}
```

```
package edu.finki.np.av2.bank;

public class Bank {
    private Account[] accounts;
    private int totalAccounts;
    private int max;

    public Bank(int max) {
        this.totalAccounts = 0;
        this.max = max;
        accounts = new Account[max];
    }

    public void addAccount(Account account) {
        if (totalAccounts == accounts.length) {
            Account[] old = accounts;
            max *= 2;
            accounts = new Account[max];
            for (int i = 0; i < old.length; i++) {
                accounts[i] = old[i];
            }
        }
        accounts[totalAccounts++] = account;
    }

    public double totalAssets() {
        double sum = 0;
        for (Account account : accounts) {
            sum += account.getCurrentAmount();
        }
        return sum;
    }

    public void addInterest() {
        for (Account account : accounts) {
            if (account instanceof InterestBearingAccount) {
                InterestBearingAccount iba = (
                    InterestBearingAccount) account;
                iba.addInterest();
            }
        }
    }
}
```

```
package edu.finki.np.av2.bank;
```

```
public class InterestCheckingAccount extends Account implements
    InterestBearingAccount {

    public static final double INTEREST_RATE = .03; // 3%

    public InterestCheckingAccount(String holderName, int number
        ,
            double currentAmount) {
        super(holderName, number, currentAmount);
    }

    @Override
    public void addInterest() {
        addAmount(getCurrentAmount() * INTEREST_RATE);
    }

}
```

```
package edu.finki.np.av2.bank;

public class PlatinumCheckingAccount extends Account implements
        InterestBearingAccount {

    public PlatinumCheckingAccount(String holderName, int number
        ,
            double currentAmount) {
        super(holderName, number, currentAmount);
        // TODO Auto-generated constructor stub
    }

    @Override
    public void addInterest() {
        addAmount(getCurrentAmount() * InterestCheckingAccount.
            INTEREST_RATE
                * 2);
    }

}
```