

Pretpostavite da postoji sučelje `ReceivingSystem` koje specificira ponašanje sustava za generiranje redoslijeda u nekoj ustanovi. Sučelje će biti objašnjeno u nastavku.

Vaš je zadatak implementirati razred `BankReceivingSystem` koji implementira navedeno sučelje i specificira generiranje redoslijeda u banci.

```
interface ReceivingSystem{
    public void customerArrived(boolean urgent);
    public void customerLeft(boolean urgent);
    public void calculateEmployeeStatus();
    public int getUrgentListSize();
    public int getNonUrgentListSize();
    public boolean isEmployeeOccupied();
    public int getNumberOfArrivedCustomers();
}
```

Pojašnjenje sučelja:

- `public void customerArrived(boolean urgent);`
 - Korisnika se pri ulasku u banku tretira prema hitnosti (hitni i ne hitni).
 - **@param urgent** definira hitnost korisnika.
 - Ako je njena vrijednost istina, korisnik se smatra hitnim korisnikom i povećava se broj ljudi koji čekaju u redu za hitne korisnike. Ako je neistina, vrijedi obrnuto (korisnik nije hitan i povećava se broj ljudi koji čekaju u redu za ne hitne korisnike).
- `public int getUrgentListSize();`
 - **@return** Vraća broj korisnika u **HITNOM** redu.
- `public int getNonUrgentListSize();`
 - **@return** Vraća broj korisnika u **NE HITNOM** redu.
- `public void customerLeft(boolean urgent);`
 - Poziv sljedeće metode označava da je korisnik napustio banku i broj ljudi u odgovarajućem redu se smanjio. Zaposlenik na šalteru je sada slobodan.
 - **@param urgent** određuje koji red će brojati jednog člana manje.
 - Ako je vrijednost istina, korisnik je bio hitan i treba se smanjiti broj ljudi u redu za hitne korisnike. Inače, obrnuto.
- `public void calculateEmployeeStatus();`
 - Poziva se svaki puta kada korisnik stigne ili kada korisnik napusti banku.
 - Svrha ove metode je odrediti status zaposlenika (ako ima korisnika u redovima zaposlenik će postati zauzet - vrijednost odgovarajuće varijable je istina, inače zaposlenik će ostati slobodan - vrijednost odgovarajuće varijable je neistina).

- `public boolean isEmployeeOccupied();`
 - **@return** Vraća boolean vrijednost je li korisnik zauzet. Istina znači da je zauzet, neistina da je slobodan.
- `public int getNumberOfArrivedCustomers();`
 - **@return** Vraća ukupni broj korisnika koji je stigao u banku.

Konstruktor razreda: Ne prima nikakve argumente. Početno stanje zaposlenika je slobodan, veličine redova u banci su na nuli i brojač pristiglih korisnika u banci je na nuli.

Opaska: Razred definirajte na razini package-private (dakle, bez modifikatora vidljivosti). Ako trebate dodati članske varijable, one moraju imati minimalnu vidljivost.

Primjer scenarija:

HITNI korisnik stiže u banku - broj hitnih korisnika se povećava za 1.

Zaposlenik banke je na početku slobodan - hitni korisnik može doći obaviti željni posao - zaposlenik banke je sada zauzet.

Dolazi novi NE HITNI korisnik u banku - povećava se broj NE HITNIH korisnika za jedan.

Dolazi novi HITNI korisnik - broj hitnih korisnika se povećava za 1.

Broj HITNIH korisnika je 2 - jer je prvi korisnik još uvijek na šalteru, a drugi je sada došao u banku.

Broj NE HITNIH korisnika je 1.

HITNI korisnik odlazi iz banke - broj hitnih korisnika se smanjuje - zaposlenik je slobodan - drugi HITNI korisnik dolazi na red - zaposlenik je zauzet.

Broj HITNIH korisnika je 1 - jer je korisnik trenutno na šalteru.

Broj NE HITNIH korisnika je 1.

HITNI korisnik odlazi iz banke - smanjuje se broj hitnih korisnika i sada iznosi 0 (nema nikoga u redu čekanja) - zaposlenik je slobodan - korisnik iz NE HITNOG reda dolazi na red - zaposlenik je zauzet.

Broj NE HITNIH korisnika je i dalje 1.

NE HITNI korisnik odlazi iz banke - smanjuje se broj NE HITNIH korisnika - zaposlenik je slobodan.

Ukupni broj korisnika koji je stigao u banku je 3.

Primjer izvođenja:

```
BankReceivingSystem brs = new BankReceivingSystem();

brs.customerArrived(true);
System.out.println(brs.isEmployeeOccupied());
brs.customerArrived(false);
brs.customerArrived(true);

System.out.println(brs.getUrgentListSize());
System.out.println(brs.getNonUrgentListSize());

brs.customerLeft(true);
System.out.println(brs.isEmployeeOccupied());
System.out.println(brs.getUrgentListSize());
System.out.println(brs.getNonUrgentListSize());

brs.customerLeft(true);
System.out.println(brs.isEmployeeOccupied());
System.out.println(brs.getUrgentListSize());
System.out.println(brs.getNonUrgentListSize());

brs.customerLeft(false);
System.out.println(brs.getNonUrgentListSize());
System.out.println(brs.isEmployeeOccupied());

System.out.println(brs.getNumberOfArrivedCustomers());
```

Izlaz:

```
true
2
1
true
1
1
```

lzlaz:

true

2

1

true

1

1

true

0

1

0

false

3

1

2

Napisati klasu `Block` koja predstavlja blok transakcija u lancu blokova (engl. blockchain). Ova klasa ima atribut `prevHash` koji predstavlja *hash* prethodnog bloka u lancu blokova te njegov *getter* `getPrevHash` i *setter* `setPrevHash(byte[] prevHash)`. Transakcije pohranjuje u polju objekata tipa `String` te stoga treba imati metodu `int add(String transaction)` za dodavanje nove transakcije u polje i metodu `void remove(int index)` za brisanje transakcije s određenim indeksom iz polja, pri čemu prva metoda vraća indeks polja u koje je dodana transakcija. Transakcije bi trebalo slijedno dodavati u polje: prva koja se doda bi trebala imati indeks `0`, sljedeća indeks `1`, itd. Kad se neka transakcija s indeksom `index` obriše iz polja tada bi trebalo sve transakcije s indeksom `>=index` pomaknuti ulijevo da se popuni novonastala praznina u polju.

Osim toga klasa `Block` treba imati i metodu `hash` kojom se računa *hash* nekog bloka na osnovu transakcija pohranjenih u njemu i *hasha* prethodnog bloka `prevHash`. Metodu `hash` treba implementirati na sljedeći način:

```
public byte[] hash(byte[] prevHash) {
    return new SHAHasher().hash(prevHash, this.transactions);
}
```

gdje je `SHAHasher` neka klasa koja implementira sučelje `Hasher`. Kod te klase i sučelja (koje **ne treba implementirati**) dan je u nastavku:

```
public class SHAHasher implements Hasher {
    public byte[] hash(byte[] prevHash, String[] transactions) {
        try {
            MessageDigest digest = MessageDigest.getInstance("SHA-256");
            byte[] transactionBytes = Arrays.toString(transactions).getBytes(StandardCharsets.UTF_8);
            byte[] data = new byte[prevHash.length + transactionBytes.length];
            System.arraycopy(prevHash, 0, data, 0, prevHash.length);
            System.arraycopy(transactionBytes, 0, data, prevHash.length, transactionBytes.length);
            return digest.digest(Arrays.toString(data).getBytes(StandardCharsets.UTF_8));
        } catch (NoSuchAlgorithmException ex) {
            System.exit(1); //this should never happened
            return null;
        }
    }
}

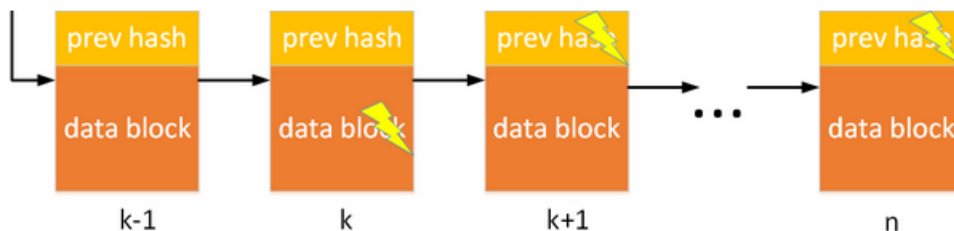
public interface Hasher {
    public byte[] hash(byte[] prevHash, String[] transactions);
}
```

Nakon toga treba napisati klasu `Blockchain` koja predstavlja lanac blokova koji je pohranjen u njegovom atributu `blocks` tipa `Block[]`. Ova klasa treba imati metodu `int add(Block newBlock)` za dodavanje novog bloka u lanac i metodu `Block get(int index)` za dohvaćanje bloka s odgovarajućim indeksom iz lanca, pri čemu prva metoda vraća indeks polja u koje je dodan blok. Prilikom dodavanja bloka u lanac potrebno je postaviti vrijednost njegovog atributa `prevHash` na

- vrijednost `new byte[]{0}` kada se radi o prvom bloku u lancu, a inače na
- vrijednost `hasha` prethodnog bloka u lancu (koja je rezultat poziva metode `hash` nad prethodnim blokom u lancu, a ne vrijednost njegovog atributa `prevHash`).

Osim toga u klasi `Blockchain` napišite metodu `boolean isAltered(int blockIndex, byte[] expectedHash)` čiji prvi argument `blockIndex` predstavlja index nekog bloka u lancu, a drugi argument `expectedHash` predstavlja očekivani `hash` tog bloka. Ova metoda treba utvrditi jesu li blokovi u lancu s indeksima $\leq \text{blockIndex}$ naknadno mijenjeni. Naknadna izmjena može biti promjena (dodavanje ili brisanje) transakcija u bloku i/ili mijenjanje vrijednosti atributa `prevHash`. Ovu metodu treba implementirati na način da prođe kroz cijeli lanac blokova od indeksa `0` do indeksa `blockIndex` te za svaki blok ponovno izračuna `hash` prethodnog bloka u lancu (koji je u svakom bloku pohranjen kao atribut `prevHash`).

Promjena transakcija u nekom bloku se može prepoznati po tome što `prevHash` vrijednosti blokova sljedbenika u lancu neće biti ispravne (tj. neće biti jednake nanovo izračunatim `hash` vrijednostima), kao što je prikazano na sljedećoj slici:




Promjena atributa `prevHash` u nekom bloku se može prepoznati po tome što tada ta vrijednost neće odgovarati nanovo izračunatoj `hash` vrijednosti prethodnog bloka u lancu.

Promjena bloka s indeksom `blockIndex` se može prepoznati po tome što tada predani argument `expectedHash` (koji predstavlja očekivanu vrijednost `hasha` tog bloka) neće biti jednak njegovoj ponovno izračunatoj `hash` vrijednosti. **Napomena:** Dva `hasha` tipa `byte[]` usporedite na sljedeći način: `Arrays.equals(firstHash, secondHash)`.

Primjer isječka koda metode `main` je sljedeći:

```
Blockchain sbc = new Blockchain(100);
Block firstBlock = new Block(10);
firstBlock.add("some transaction 1");
firstBlock.add("some transaction 2");
sbc.add(firstBlock);
byte[] firstBlockHash = firstBlock.hash(new byte[]{0});
System.out.println(sbc.isAltered(0, firstBlockHash)); //false
```

 Lightshot

Lightshot

Screenshot is saved to lab
open in the folder.

```
Block firstBlock = new Block(10);
firstBlock.add("some transaction 1");
firstBlock.add("some transaction 2");
sbc.add(firstBlock);
byte[] firstBlockHash = firstBlock.hash(new byte[]{0});
System.out.println(sbc.isAltered(0, firstBlockHash)); //false
```

```
Block secondBlock = new Block(10);
secondBlock.add("some transaction 3");
secondBlock.add("some transaction 4");
sbc.add(secondBlock);
byte[] secondBlockHash = secondBlock.hash(firstBlockHash);
System.out.println(sbc.isAltered(1, secondBlockHash)); //false
```

```
//check remove transaction in a previous block
firstBlock.remove(1);
System.out.println(sbc.isAltered(1, secondBlockHash)); //true
firstBlock.add("some transaction 2"); //return removed transaction
System.out.println(sbc.isAltered(1, secondBlockHash)); //false
```

```
//check modification of the block with index blockIndex
secondBlock.add("some transaction 5");
System.out.println(sbc.isAltered(1, secondBlockHash)); //true
secondBlock.remove(2); //remove added transaction
System.out.println(sbc.isAltered(1, secondBlockHash)); //false
```

```
//check add transaction in a previous block
firstBlock.add("some additional transaction");
System.out.println(sbc.isAltered(1, secondBlockHash)); //true
```

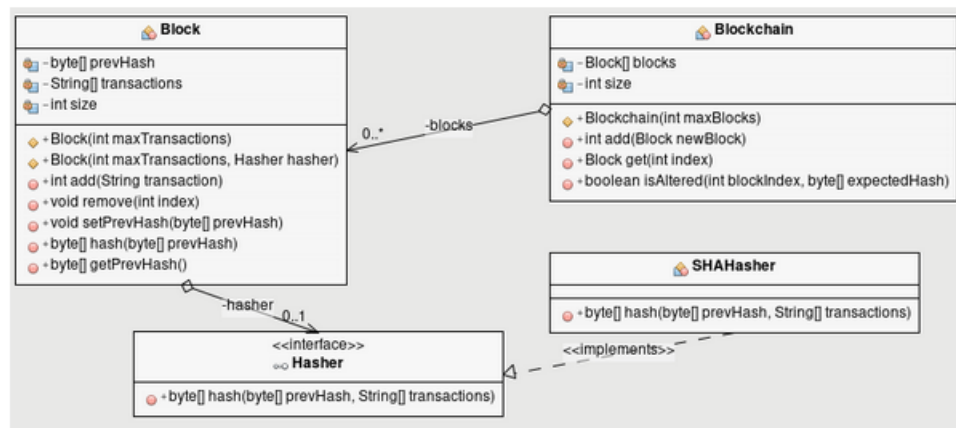
```
//check prevHash modification
byte[] alteredFirstBlockHash = firstBlock.hash(new byte[]{0});
secondBlock.setPrevHash(alteredFirstBlockHash);
System.out.println(sbc.isAltered(1, secondBlockHash)); //true
```


Primijetite da klase `Block` i `Blockchain` u konstruktorima primaju argumente tipa `int` koji predstavljaju maksimalni broj transakcija i blokova koji se mogu pohraniti u njihove odgovarajuće atribute.

Na kraju modificirajte klasu `Block` da umjesto *hashera* `SHAHasher` može podržati i druge *hashere* koji implementiraju sljedeće sučelje `Hasher`. Pri tome napravite novi konstruktor klase `Block` koji kao drugi argument može primiti referencu na *hashera*. Pri tome ostavite da stari konstruktor `new Block(int maxTransactions)` postavlja `SHAHasher` kao *hashera*. Primjer poziva jednog i drugog konstruktora je dan u nastavku:

```
Block firstBlock = new Block(10);//SHAHasher
Block secondBlock = new Block(10, new MD5Hasher());//MD5Hasher (implements Hasher)
```

Napomena: Klase `Block` i `BlockChain` napisati bez modifikatora vidljivosti i kopirati u prostor za rješenje bez navođenja paketa kojem pripadaju. Za obje klase nije potrebno provjeravati jesu li predane ispravne vrijednosti argumenata. Dijagram klasa je prikazan na sljedećoj slici:



Zadan je razred:

```
class Student {  
    private String name, surname;  
    private int age;  
    public Student(String name, String surname, int age) {  
        super();  
        this.name = name;  
        this.surname = surname;  
        this.age = age;  
    }  
    public String getName() {  
        return name;  
    }  
    public String getSurname() {  
        return surname;  
    }  
    public int getAge() {  
        return age;  
    }  
}
```

Vaš je zadatak napisati metodu

```
static void sortStudentsOnAge(Student[] students)
```

kojom se sortira ulazno polje tipa Student uzlazno prema godinama studenata.

*Napomena: kao rješenje se unosi samo gore navedena metoda, ne i glavni program

**Napomena: manipulacije odnosno sortiranje je potrebno napraviti nad predanim poljem.

***za sortiranje možete koristiti bilo koji postojeći algoritam (prijedlog: <https://www.javatpoint.com/bubble-sort-in-java>)

 Lightshot

Lightshot

Screenshot is saved to lab24.png. Click
open in the folder.