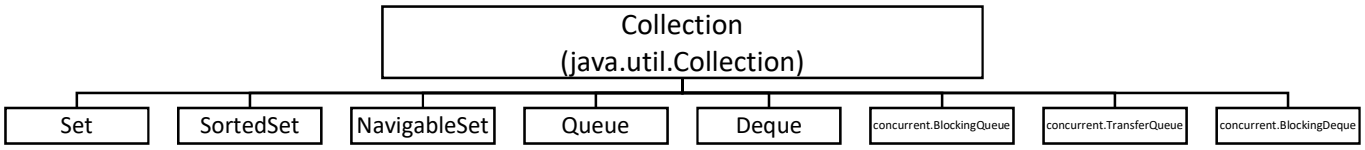
Рамка Collections

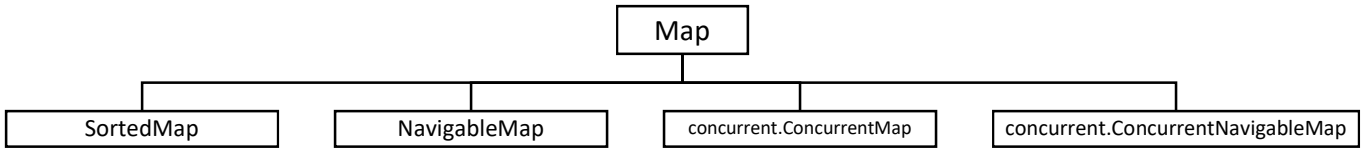
Платформата на Java включва в себе си една *рамка колекции* (*collections framework*). Колекцията (*collection*) е обект, който сам по себе си представлява набор от други обекти. ***Рамката** колекции* е стандартизирана структура за работа с колекции, която позволява те да се обработват, без да се вниква в дълбочина на алгоритмите, с които това се извършва. Грубо можем да си представим рамката като „език в езика“, чието ползване повишава многократно ефективността на програмистката работа, като едновременно се подобряват и качествата на разработвания продукт. Рамката е един вид *идеология*, по което се различава от библиотеките, които можем да разглеждаме като *инструментариум*. С идеологията *се съобразяваме*, а инструментите *използваме*. Т. е., в някаква степен контролът е обърнат: рамката *ни задължава* да спазваме идеологията, библиотеките са просто готов код, който можем да *използваме* (адекватно или не). И единият, и другият вид средства, които разработваме или използваме, имат своето място в програмирането.

Рамката Collections включва различни видове интерфейсни колекции като динамичен масив, списък, асоциативен масив, множество и т. н. Те очаквано са подредени в йерархии. Най-основният интерфейс е Collection.



(Когато искаме да използваме някоя от структурите, включваме съответната „библиотека“ с `import java.util.`*`име_на_структурата`*`;`)

Друга йерархия от интерфейси наследяват асоциативна структура **Map** (`java.util.Map`):



По-нататък ще обърнем внимание на специалните изисквания (теоретичната идеология) при работа с колекциите. Сега ще се обърнем към конкретни приложения на рамката Collections:

Интерфейс	Хеш-таблица	Динамичен масив	Балансирано дърво	Свързан списък	Свързан спсък с хеш-таблица
`Set`	`HashSet`		`TreeSet`		`LinkedHashSet`
`List`		`ArrayList`		`LinkedList`	
`Deque`		`ArrayDeque`		`LinkedList`	
`Map`	`HashMap`		`TreeMap`		`LinkedHashMap`

Динамичен масив

Ние можем да реализираме динамичен масив (от конкретен тип), ползвайки стандартен масив, който разширяваме „при нужда“. Ако искаме обаче да го направим обобщено (в шаблонен вид), нещата няма да са така прости. За всеки нов обект ще се наложи да вземаме памет от динамичната, а масивът да е само от указатели към тези обекти. Не че не можем да го постигнем, но всичко може да стане с много по-малко усилия (и грешки), ако ползваме рамката Collections и по-точно – приложението ѝ **ArrayList**.

Задача. Да се създаде клас Student (име, оценка_по_информатика). Да се позволи въвеждане от стандартния вход на обекти от този клас, до въвеждане на празен низ за име. Да се изведе въведената информация, като всеки ред съдържа един елемент във вида *име*: *оценка_с_думи* (*оценка_с_цифра*). На последния изходен ред да се изведе средната аритметична оценка на въведените ученици.

Структура на входа:

- нечетен ред: име (може да съдържа интервали)
- четен ред: оценка (цяло число от 2 до 6)
- празен нечетен ред - край на входа

```java
import java.util.Scanner;
public class Student {
 private String name;
 private byte mark;
 public Student(Scanner inp){//Конструктор от входен поток
 /*Структура на входа:
  *ред 1: име (може да съдържа интервали)
  * (празен низ за име - специален обект)
  *ред 2: оценка (цяло число от 2 до 6)
  */
  name=inp.nextLine(); //Вход на цял ред за име
  if (!name.equals("")) {//Ако името не е празният низ...
   mark=inp.nextByte();  //... четене на оценката,
   inp.nextLine();       //... и изчистване на входния буфер
  }
 }
 private String words(){//Оценка с думи
  switch(mark){
   case 2:return "poor";
   case 3:return "satisfactory";
   case 4:return "good";
   case 5:return "very good";
   case 6:return "excellent";
  }
  return "";
 }
 //Гетъри
 public String getName(){
  return name;
 }
 public int getMark(){
  return mark;
 }
 //Коректност на оценката
 public boolean correctMark(){
  return mark>=2 && mark<=6;
 }
 @Override
 public String toString(){
  return name+": "+words()+" ("+mark+")";
 }
}
```

```java
import java.util.Scanner; //За входен поток
import java.io.File; //За работа с файков вход
import java.util.ArrayList; //Динамичен масив
public class Problem {
 //Симулация на "условно компилиране":
 // ако FILEINPUT е true, входът е от файл,
 // иначе е от стандартния вход.
 public static final boolean FILEINPUT=false;
 static ArrayList<Student> Lst=new ArrayList<Student>();
 public static void main(String[] args) {
  Student s;//Елемент от тип Student
  int c=0;//суматор за оценките
  File file;
  Scanner inp;
  if (FILEINPUT) file=new File("Data.txt");//За вход от файл (в папката на проекта)
  else file=null;
  try{
   if (FILEINPUT) inp=new Scanner(file); //При вход от файл
   else inp=new Scanner(System.in);
   do{
     s=new Student(inp);//Нов елемет от потока
     if (s.getName().equals("")) break; //Проверка за край на входа
     if (s.correctMark()) { //Ако е коректно,
         Lst.add(s);        //добавяне към динамичния масив.
         c+=s.getMark();    //Натрупване на числата
     }
   }while(true);//Псевдобезкраен цикъл
   inp.close(); //Затваряне на входния поток
   //Обхождане на масива за извеждане на информация
   for (Student t:Lst) System.out.println(t.toString());
   //Средно аритметично
   if (Lst.size()>0) System.out.println("Average: "+(double)c/Lst.size());
   else System.out.println("No data.");
  }catch(Exception e){System.out.println("File not found");}
 }
}
```

Нека допълним задачата с изискването списъкът да е нареден по някакъв начин, например: учениците да се извеждат намаляващо по оценките си, а тези с еднакви оценки – растящо по азбучен ред на имената.
Решението, което можем да направим без използване на Collections-рамката, е от този характер:

1. Ще е необходим инструмент за размяна на свойствата на два обекта от тип Student. Нормално е, както досега, да реализираме някак метод swap в класа Student:

```
public void swap(Student a){
 String t=name;
 name=a.name;
 a.name=t;
 byte m=mark;
 mark=a.mark;
 a.mark=m;
}
```

2. Ще трябва да реализираме някой от познатите ни алгоритми за сортиране в класа Problem, например чрез пряк избор (Select sort):

```
private static int minIndex(int start){//метод за намиране на индекса на "най-малкия"
                                       //елемент надясно от номер start (включително)
 int m=start;
 for (int i=start+1;i<Lst.size();i++){
  if (Lst.get(i).getMark()>Lst.get(m).getMark()) m=i;//по-висока оценка
  else if (Lst.get(i).getMark()==Lst.get(m).getMark() && //при еднакви оценки
            Lst.get(i).getName().compareTo(Lst.get(m).getName())<0) m=i;//прав
                                                                    //азбучен ред
 }
 return m;
}
public static void selSort(){ //Сортиране
 for (int i=0;i<Lst.size()-1;i++) Lst.get(i).swap(Lst.get(minIndex(i)));
}
```

Сега е достатъчно в `main()` да повикаме `selSort()` преди извеждане на резултатите.
Такова решение е универсално и винаги приложимо, но изисква наново и наново да реализираме някой от методите за сортиране (за предпочитане е, разбира се, някой с по-малка сложност, например „бързо сортиране“). След запознаване с възможностите на шаблоните (generics), много ни се иска това да бъде избегнато. Даже още повече – ако може сортирането (един често използван инструмент) да бъде част от готова библиотека. Ако се замислим, обаче, ще схванем, че няма как да се напише такава библиотека, валидна за всички класове (даже и тези, които тепърва ще се реализират): няма как да знаем предварително какво за дизайнера на нов клас означава „по-малко“ („по-добро“, „по-напред“). Тук ще дойде на помощ **идеологията**, т. е. – **рамката** Collections. Това е много подходящо място да се разбере с конкретен пример и разликата между библиотека и рамка, „обърнатият контрол“, за който стана дума. Да, Collections има инструмент за сортиране и той, очаквано, се нарича `sort`. Но за да бъде той приложим, **рамката изисква** от разработчика да подчини класа си на съответен дизайн.
Един от възможните дизайни е класът да имплементира интерфейс Comparable. В конкретния случай:

```
public class Student implements Comparable<Student>
```

От своя страна, този интерфейс **изисква** от разработчика реализация на метод `compareTo`, който има за параметър обект от същия клас и връща цяло число (**`int`**). При това, **идеологията** включва следното: ако `this` според разработчика е „по-хубаво“ („трябва да е по-напред“) от обекта-параметър, връщаният резултат трябва да е отрицателен (най-често -1). Ако, напротив, параметърът е „по-хубав“ от `this`, резултатът трябва да е положителен (например 1). И последно, ако двата обекта са неразличими (т. е., според критериите на разработчика е все едно кой ще е „по-напред“), връщаният резултат трябва да е нула. В нашия пример:

```
@Override
public int compareTo(Student s){
 if (mark>s.mark) return -1;
 if (mark<s.mark) return 1;
 return name.compareTo(s.name);
}
```

Забелязвате, че накрая използваме `compareTo` за низовете-имена: в класа String този метод е реализиран и, както сега можем да заключим, логиката му точно отговаря на идеологията. Както можете да се досетите, всички стандартни типове имплементират Comparable и, следователно, ако ArrayList е върху някой от тях, сортирането е директно приложимо и *подредбата е в ненамаляващ ред*.
Остава вместо нашия метод `sort()`, в **main** да повикаме `Collections.sort(Lst);` и резултатът ще е налице. Без `swap()`, без `minIndex()` и без `sort()` – само неизбежният код за `compareTo()`.

Какво става, ако потрябват ***и други*** подредби на елементите в ArrayList? За целта можем да ползваме библиотеката `java.util.Comparator` и да създадем клас, който имплементира интерфейс `Comparator` върху нашия клас (Student). Този интерфейс **изисква** реализиране на метод `compare`, който приема като параметър два обекта от обобщения (generic) тип и връща **int** точно по същата идеология, описана по-горе за интерфейса `Comparable`. Ето как може да изглежда такъв клас в нашия проект, който е предвиден за подредба, нарастваща по оценките, а при еднакви оценки – намаляваща по азбучен ред:

```
import java.util.Comparator;
public class Cmp implements Comparator<Student> {
 public int compare(Student a, Student b){
  if (a.getMark()<b.getMark()) return -1;
  if (a.getMark()>b.getMark()) return 1;
  return b.getName().compareTo(a.getName());
 }
}
```

Сега в **main** можем да повикаме `Collections.sort(Lst,new Cmp());` (с втори параметър – обект от класа, който осигурява информация за „по-хубав“ от два елемента) или дори просто `Lst.sort(new Cmp());`, защото метод `sort()` е предефиниран и в класа ArrayList с точно такъв „подреждащ“ параметър. Разбира се, най-големият плюс на подхода с `Comparator` е, че може да си направим колкото искаме такива „подреждащи“ класчета като `Cmp` и лесно да подреждаме един ArrayList по различни начини.

Задача

Казваме, че цялото положително число X е „по-унарно“ от цялото положително число Y:

- ако в двоичния запис на X цифрата 1 се съдържа повече пъти, отколкото в двоичния запис на Y;
- ако броят на единиците в двата десетични записа е един и същ, „по-унарно“ е по-голямото от тях.

Напишете конзолно приложение на Java, при което:

1. От стандартния вход се въвежда редица R от цели положителни числа, всяко с не повече от 18 десетични цифри, разделени с точно един интервал. Входът е коректен и приключва с числото нула.
2. На стандартния изход се извежда един ред с редицата R, като числата в него са подредени намаляващо по „унарност“ и са разделени с интервал.

ПРИМЕР:

ВХОД	**ИЗХОД**	***Обяснение***
`1024 127 33 56 42 777 35 40 0`	`127 777 56 42 35 40 33 1024`	$1024=10000000000_2$ $127=1111111_2$ $33=100001_2$ $56=111000_2$ $42=101010_2$ $777=1100001001_2$ $35=100011_2$ $40=101000_2$