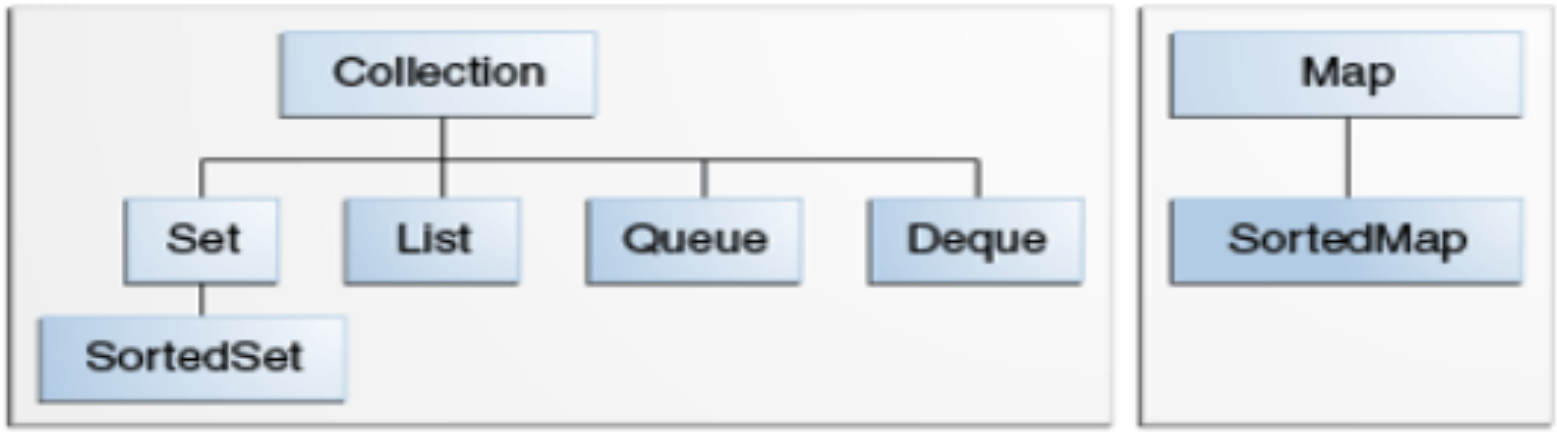
Streams

Gränssnittet Collection

- Beskriver egenskaper som är gemensamma för alla samlingsklasser
- Elementen i en, List, Queue och Deque ligger i en ordnad sekvens
- Elementen i ett Set (=mängd) har ingen inbördes ordning



The core collection interfaces.

Gränssnittet Collection

- List <>
  - Den gamla vanliga, inga konstigheter
- Queue <>
  - Element läggs in sist i kön
  - När element tas ut plockas de frön början av kön
- Deque <>
  - Double ended queue
  - Kan lägga in element både i början och i slutet
  - Kan plocka element både från början och slutet
- Set<>
  - En oordnad mängd av element
  - Vi kan inte använda oss av index för att komma åt ett visst element

Collections, implementerande klasser

- List
  - ArrayList, LinkedList, Vector
  - Implementerar Serializable
- Queue
  - LinkedBlockingQueue, ArrayBlockingQueue, PriorityBlockingQueue
- Deque
  - BlockingDeque, ArrayBlockingDeque
- Set
  - HashSet, TreeSet, EnumSet
  - Implementerar Serializable

Collection, användbara metoder

- add(e) - lägger in ett element
- addAll(s) – lägger in alla element från collection c
- clear()
- contains(o) – ingår ett objektet o i denna collection?
- isEmpty() – är denna collection tom?
- size() – hur många element finns i denna collection
- toArray() – Gör om samligen till en array
- forEach(c) – utför operationen c på alla element
- removeIf(p) – tar bort ett element om predikatet p är uppfyllt
- stream() – ger en ström som hämtar data från samlingen

Collection, användbara metoder, forts

- Vissa metoder har funktionsgränssnitt som parametrar
- De kan anropas med lambda-uttryck
  - l.forEach(s -> System.out.println(s));
    - För varje element i samlingen, gör det som står efter ->
  - l.removeIf(s -> s.equals(”Not used”));
    - För varje element i samlingen som uppfyller kravet efter ->, ta bort det elementet
- Notera att gränssnittet Collection inte är samma sak som klassen Collections som innehåller statiska hjälpmetoder som kan användas på samlingsklasser

Arrays

- Precis som det finns en klass Collections med statiska metoder för samlingar finns det en klass Arrays med statiska hjälpmetoder för arrayer
- En mycket användbar metod där är Arrays.asList(e1, e2, e3);
- Den skapar en lista av med elementen e1, e2, e3
- Vi behöver inte göra .add() för varje element vi lägger in i listan.

Strömmar

- Ett otroligt kraftfullt stöd att manipulera stora mängder data snabbt och effektivt
- Vi anropar metoden stream() för en samling och plötsligt har vi tillgång till en mängd olika metoder för datamanipulation att göra på elementen i samlingen
- Enkelt exempel (antag att l är en lista av typ List<String>):
  - l.stream().forEach(i -> System.out.println(i));
- Ovanstående skriver ut alla strängar i l
- Får bara innehålla referensvariabler
  - Om primitiva variabler ska finnas i strömmen, använd klasserna IntStream, DoubleStream eller LongStream

- Strömmar öppnar för funktionell programmering i Java sedan version 8
- Fördelar med strömmar:
  - Gör dig mer effektiv
  - Använder sig av lambdauttryck
  - ParallellStreams gör det enkelt att jobba med flera trådar (mer om det senare)
- En ström består av:
  - En källa (source)
  - Noll eller fler mellanoperationer
  - En avslutande operation

- Stream source (källa)
  - Kan skapas från Collections, Lists, Sets, ints, longs, doubles, arrays, lines of a file
- Stream operations (operationer)
  - Intermediate (mellanliggande) operations såsom filter, map eller sort returnerar en en ström så vi kan chaina (kedja ihop) flera operationer
  - Terminal (slutliga) operations såsom forEach, collect eller reduce är antingen void eller returnerar något som inte är en stream.

Intermediate operations

- Noll eller flera är tillåtna
- Om man har stora dataset spelar ordningen roll: filtrera först, sedan sortera eller mappa
- För väldigt stora dataset kan man använda ParalellStream för att använda flera trådar. (Mer om trådar senare.)
- Några intermediate operations:

anyMatch()	distinct()	filter()
flatmap()	map()	skip()
findFirst()	sorted()	

Terminal operations

- En är tillåten
- forEach() kör samma funktion på varje element
- collect sparar alla värden till en Collection
- Andra ”summerar” ihop värdena till ett värde
  - count()
  - min()
  - max()
  - reduce()
  - summaryStatistics()

Dags att kolla exempel

Filter (mellanliggande funktion)

- L är en lista av typ List<String>
- ```
  L.stream().filter(s -> s.length() > 10)
  .forEach(s -> System.out.println(s));
  ```
- Filter(p) tar ett lambda-uttryck av typen Predicate som parameter
  - Predicate är ett funktionsgränssnitt som innehåller den abstrakta metoden test(T t) som returnerar en boolean.
- Ovanstående uttryck skriver ut alla ord i L som är längre än 10 tecken
- Det går bra att koppla ihop hur många mellanliggande operationer som helst efter varandra
  ```

Map (mellanliggande funktion)

- L är en lista av typ List<String>
- ```
  L.stream().filter(s -> s.length() > 10)
    .map(s -> s.toUpperCase())
    .forEach(s -> System.out.println(s));
  ```
- Map(f) tar ett lambda-uttryck av typen Function som parameter
  - Function är ett funktionsgränssnitt som innehåller den abstrakta metoden apply(T t) som returnerar ett objekt av samma typ som skickades in.
- Ovanstående uttryck skriver ut alla ord i L som är längre än 10 tecken, omgjorda till versaler
  ```

FlatMap

- Plattar ut en samling
- Om vi har listor av listor gör flatMap att alla elementen skrivs ut på en lista
- Antag att vi har en lista l av typ List<List<String>> som ser ut enligt följande: [[“hej”, “på”, “dig”],[“eller”, “inte”]]
- `l.stream().flatMap(value -> value.stream()).collect(Collectors.toList());` ger listan [“hej”, “på”, “dig”, “eller”, “inte”]
- Notera att flatMap bara plattar ut en nivå åt gången.
- För flera nivåers utplattning kan vi anropa flatMap igen på strömmen som genereras.

Collect (avslutande funktion)

- Används när du vill omvandla din ström tillbaka till en lista
- L är en lista av typ List<String>:
- ```
  L.stream().filter(s -> s.length() > 10)
    .map(s -> s.toUpperCase())
    .collect(Collectors.toList());
  ```
- Collect(c) tar en Collector som parameter
- Ovanstående returnernar en lista med bara ord längre än 10 tecken, skrivna med versaler.
  ```

Reduce

- Reducerar värdena i en samling på något sätt
- Antag att vi har en lista l av typ List<Integer> och vi vill summera värdena i listan
- `int sum = l.stream().reduce(0, (u,e) -> u+e);`
- Nollan är startvärdet för reduceringen
- u är det hittills beräknade värdet
- e är det aktuella elementet
- Här behöver vi inte göra om slutvärdet utan reduce returnerar ett element av samma typ som elementen i samlingen som reducerades

Hantering av primitiva variabler

- Primitiva variabler kan inte ingå i en samling (bara referensvariabler kan göra detta)
- Istället finns klasserna IntStream, DoubleStream och LongStream som hanterar int, double och longs.
- För att omvandla en List<Integer> till en IntStream:
  - l.stream().mapToInt(e -> e)
- Även mapToDouble och mapToLong finns för att mappa efter behov
- intList.stream().mapToInt(Integer::intValue)

Sum(), Max(), Min()

- Ovanstående metoder finns bara för IntStream, DoubleStream och LongStream
- Ett alternativ till att använda reduce för att summera:
  - `l.stream().mapToInt(e -> e).sum();`
- Hittar det högsta värdet i en List<Double>:
  - `l.stream().mapToDouble(e -> e).max();`
- Hittar det lägsta värdet i en List<Long>:
  - `l.stream().mapToLong(e -> e).min();`

Övningsuppgift 5a (5.7.5)

- Ladda ner filen `temp.txt` från Ping Pong, den innehåller temperaturer som mätts upp kl 13 på en plats under en månad. Lägg filen i den katalog där du skriver ditt program.
- Skriv ett program som läser filen och skriver ut högsta och lägsta värden, samt beräknar medeltemperaturen för månaden.
- Använd gärna String.format eller System.out.println för att formattera utskriften på fint sätt.

Klassen File

- En abstraktion av en fil eller mapp i filsystemet
- Kan inte skriva eller läsa filer
  - Då används t.ex FileWriter eller FileReader
- Konstruktorer
  - File(String path)
  - File(String path , String filename)
  - File(File dir, String filename)

File, användbara metoder

- String getName() - filnamnet
- String getPath() – ger mapp och filnamn
- String getAbsolutePath() – ger hela sökvägen
- boolean exists() – finns det en riktig fil?
- boolean isFile()
- boolean isDirectory()
- boolean isHidden()

File, användbara metoder, forts

- boolean mkdir() – skapar den mapp som File:en representerar
- boolean createNewFile() – skapar den fil som File:en representerar
- boolean delete() – raderar den mapp/fil som File:en representerar
- boolean renameTo(File f) – byter namn på fil/mapp
- File[] listFiles(); listar filer i en mapp

- För att skapa en fil:

```
File f1 = new File("min fil");
f1.createNewFile();
```

Klassen Path

- Nästa generation fil-hanterare, står inte i boken
- I paket java.nio.file
- Gör allt som File gör – fast bättre!
  - Men ibland också lite krångligare
- Varför nämnde jag då ens File?
  - Det är en av viktigaste klasserna, historiskt sett. Att göra den obsolet skulle innebära att väldigt mycket existerande Java-kod måste skrivas om.
  - När ni börjar jobba kommer ni garanterat att stöta på File och vara tvungna att hantera File-objekt. Ta då gärna tillfället i akt och uppgradera till Path.

Path

- Annorlunda instansiering, mha hjälpklassen Paths:

```
Path p = Paths.get("filnamn");
```

- Ovanstående gör egentligen:
  - `Path p = FileSystems.getDefault().getPath("filnamn");`

Try/catch with resources

- När ett fel inträffar när vi läser från en fil finns en risk att koden hoppar iväg för att ta hand om felet, och file lämnas kvar - öppen.
- När en fil är öppen är den ”paxad” – ingen annan kommer åt den.
- För att undvika detta scenario finns try-with-resources
- Readers och Writers instansieras direkt efter try
- Java tar hand om att stänga filer automatiskt

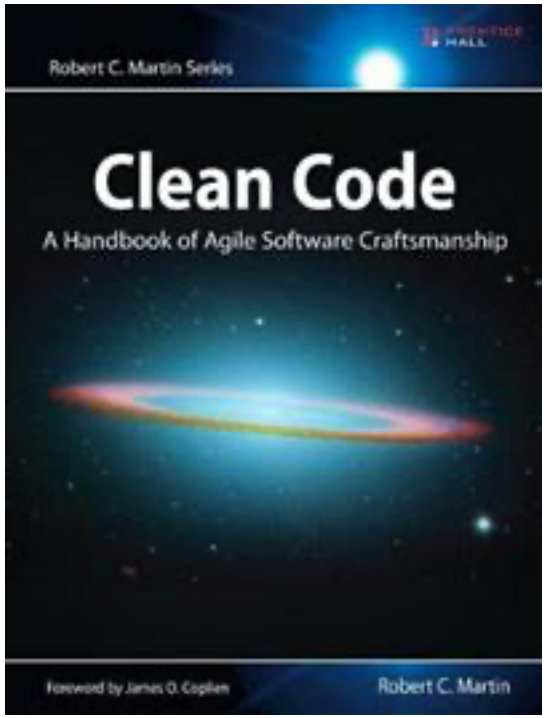
Övningsuppgift 6, felhantering (5.7.6)

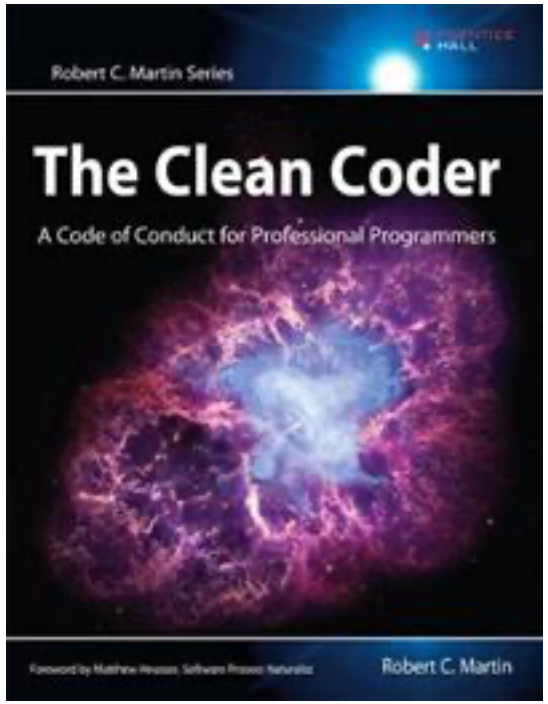
- Ladda ner filen `personuppgifter.txt` från Ping Pong.
- Filen innehåller personuppgifter. För varje person står personens namn, adress och på nästa rad personens ålder, längd och vikt. Du ska läsa in filen i ditt program och hitta alla personer som är längre än 2 meter.
- Skapa sedan en ny textfil som bara innehåller uppgifterna för de långa personerna.
- Både infilens och utfilens namn ska läsas in av programmet.
- Använd try-with-resources
- Exempel på personpost i infilen:

  Kalle Nilsson, Xvägen 1, 12345 Ystad
  25, 80, 175

Clean Code

- https://www.adlibris.com/se/bok/clean-code-9780132350884
- https://www.adlibris.com/se/bok/clean-coder-the-9780137081073





Clean Code - kommentarer

- En fin kommentar räddar inte dålig kod
  - Om koden är kass– skriv om den!
- Sträva efter att förklara dig i koden, förlita dig inte på kommentarerna
  - Meningsfullt namngivna variabler och funktioner
  - Tydlig logik
  - Stringent struktur

Clean Code – bra kommentarer

Observera att alla nedanstående punkter bara ska användas när de verkligen behövs. Helst ska koden förklara sig själv

- Information
- Förmedla avsikt
- Förtydligande
- Varna för tråkiga konsekvenser
- TODO (fast fixa hellre TODO:et om det är möjligt)
- Lyfta fram något viktigt
- Javadoc för publika API:er

Clean Code – dåliga kommantarer

- Otydliga
- Överflödiga
- Felaktiga
- Automatgenererade template-kommentarer
- Historia över hur en viss kodsnutt utvecklats
  - Allt finns i versionshanteringssystemet
- Överdrivet pratiga
- Frågor
  - Kommentarsfälten är inte rätt plats för detta
- ASCII-mönster som delar av sidan

Clean Code – dåliga kommantarer

- Utkommenterad kod
  - Så lite dödkött som bara möjligt
- Info om författare
  - det finns blame
- För mycket information
  - Ingen orkar läsa
- HTML-kommentarer
  - Inte ett problem så länge vi håller oss till Java
- Javadoc i icke-publik kod
  - Bättre att bara titta i koden

Kul tips

- Regga konto på https://www.codewars.com/
- Gör några av utmaningarna
- Tävla / hjälps åt i gruppen

Sammanfattning

- Strömmar abstraherar dataflöden
- Byteströmmar för binärer, charströmmar för text
- Klassen File är det gamla sättet att abstrahera en fil
- Klassen Path är nyare och bättre
- Hjälpklasserna Paths och Files innehåller alla upptänkliga filrelaterade hjälpmetoder.
- Bra kommentarer ursäktar inte kass kod
- Koden ska i största möjliga mån vara självförklarande